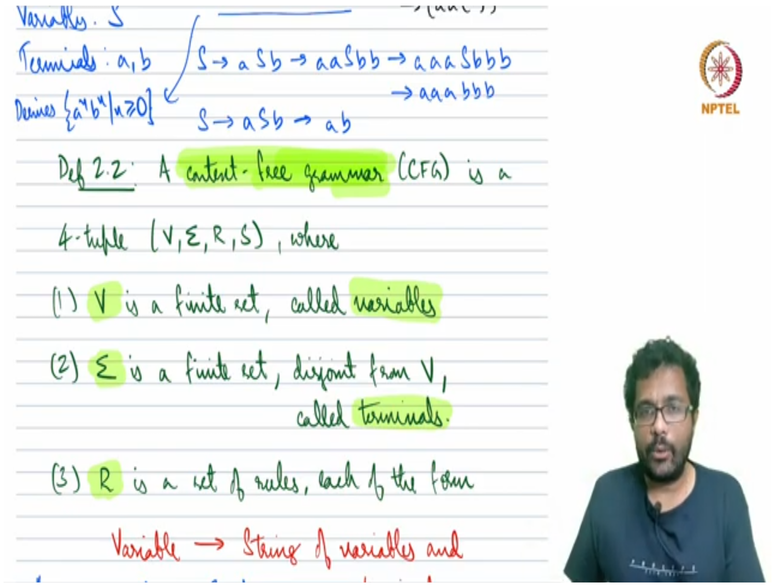
Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad
An Introduction to Context-Free Languages – Part 02

(Refer Slide Time: 00:17)



Variables: S

Terminals: a, b

Derives $\{a^n b^n \mid n \geq 0\}$

$S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$

$S \to aSb \to ab$

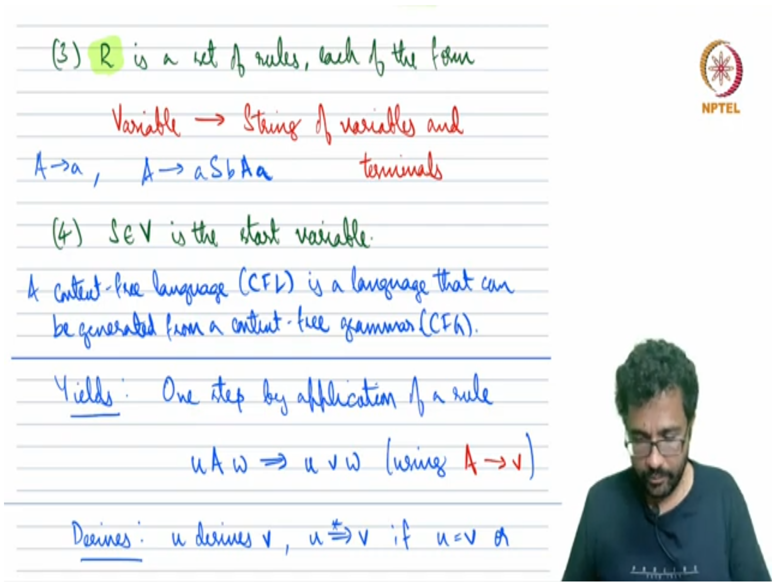
Def 2.2: A context-free grammar (CFG) is a 4-tuple (V, Σ, R, S), where

(1) V is a finite set, called variables

(2) Σ is a finite set, disjoint from V, called terminals.

(3) R is a set of rules, each of the form

Variable $\to$ String of variables and



(3) R is a set of rules, each of the form

Variable $\to$ String of variables and terminals

$A \to a$, $A \to aSbAa$

(4) $S \in V$ is the start variable.

A context-free language (CFL) is a language that can be generated from a context-free grammar (CFG).

Yields: One step by application of a rule

$uAw \Rightarrow uvw$ (using $A \to v$)

Derives: u derives v, $u \overset{*}{\Rightarrow} v$ if $u = v$ or

So, now let us see a formal definition, so much like what we saw in the case of DFAs NFAs regular expressions et-cetera. So, like always, my suggestion is to not be over awed by these definitions, but as long as you understand what is going on that is good enough. You need to know the definition but do not get lost in the notation. So, one should have an understanding of what these definitions actually say.

So, a context-free grammar is a 4-tuple. So, in DFA, we had 5-tuple; (V, Σ, R, S) where V is the set of variables. So, we already know what variables are; usually this is represented by capital letters. Σ is a finite set called terminals. And sigma is separate from V; it does not have any intersection with V.

R is a set of rules and each rule is of this form. So, you have one variable on the left side, given a string of variables and terminals on the side.

So, it could be something like for example, A → aSbAa. So, A giving a string of variables and terminals. So, it could even be something like A giving; it could even be something like A → a that is also a rule.

The string could be a very short string; so, R is a set of rules. And this R is a set of rules and what I have written here is just the two separate rules. And S is the start variable and typically the start variable. And typically the start variable is the first variable, is a the variable on the left side of the first rule that is listed.

So, you would list down the rules. You look at the first rule and see what is the variable that is that is replaced by the first rule. Or what is the variable that is featuring in the left hand side of the first rule; that is the start variable.

And one thing is, so even though I am calling its sigma; sigma is terminals, meaning they are denoted by letters, small letters, small case letters, not capital letters, or numbers like 0, 1 et-cetera. Or it could be other symbols like open bracket, close bracket, et-cetera. So, these are terminals because they cannot be further replaced by something; variables are variables, they can be replaced by something. And we do not know what rule is going to be applied.

So in that sense, we do not know what it is, what is end product. Terminal means it is, it you have terminated the processing of them and this is the end product. So, this is not going to change. So, even though we call them terminals, in some sense, the the end; in some sense, this corresponds to the alphabet that we saw in DFAs, NFAs etc.

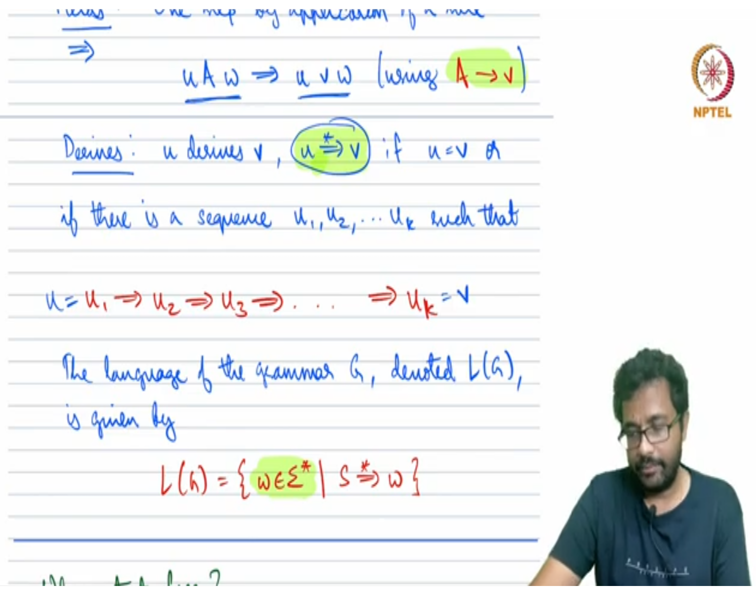
So, the string that is recognized or the string that is recognized by the DFAs would be comprising of symbols from the sigma. Similarly, in a context-free grammar, in a context-free language, all the strings will be comprising of only terminals; because this is what we are looking for. Like I do not want a string with variables in between; I want to replace all the variables with by terminals, using the rules again and again.

So finally, I will have a string of only terminals. So sigma, the string that I have is will be completely comprising of terminals, or the string will be a member of Σ^*. So, in that sense, even though in the case of DFA NFA, we call them alphabet. And here we are calling them terminals; they kind of play the same role.

And the strings that we generate in the context-free grammar is using will be part of Σ^*. So, even though I am not formally stated this; so this is a context-free grammar, do not formally written this; but a context-free language maybe I will just write it here. A context-free language is something that can be generated using context-free grammar.

A context, we will use CFL to denote a context-free language, is a language that can be generated from a context-free grammar. So, anything for which you can write a context-free grammar for is called a context-free language. So, we already seen some examples.

(Refer Slide Time: 06:37)



$\Rightarrow$

$$\underline{uAw} \Rightarrow \underline{uvw} \text{ (using } A \to v)$$

Derives: u derives v, $u \overset{*}{\Rightarrow} v$ if $u = v$ or if there is a sequence $u_1, u_2, \dots u_k$ such that

$$u = u_1 \Rightarrow u_2 \Rightarrow u_3 \Rightarrow \dots \Rightarrow u_k = v$$

The language of the grammar G, denoted $L(G)$, is given by

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

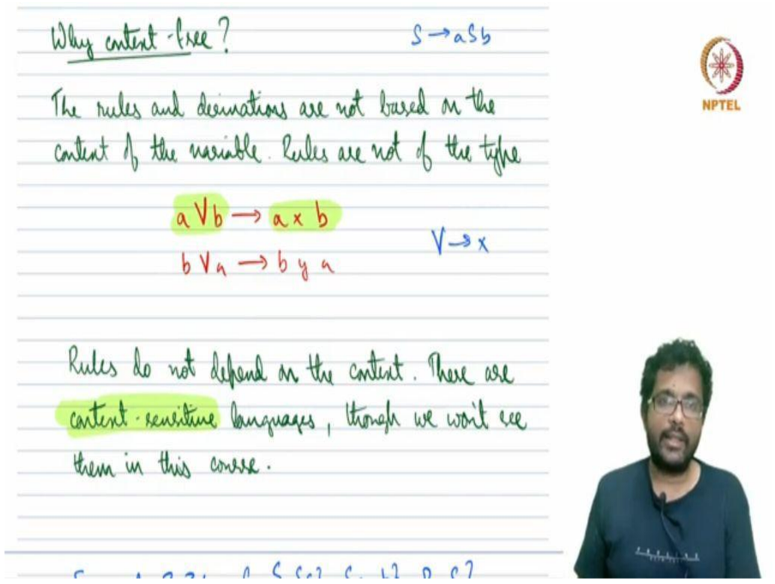
So, now let us see what, how you get a string from a grammar. So, we already seen informally, let us just see formally. So, one application of a rule is let us say you have u, A, w. So, this is a string containing one variable A; and some other thing u and some other thing w; and we are replacing A with V. So, some using some rule where A can be replaced by V. So, this is called yields; u, A, w => u, V, w. And this is denoted by the double arrow, but the double arrow. And we say derives which use similar notation, but it has a, it has a star over the arrow.

You say, u derives v. If starting from u, there is a sequence of replacements that we can get such that; so, where u is u1 and uk is v. So, u is u1; and from u1, you get u2; and from u2 yields, u2 yields, u3 yields, u4 and so on. Finally we, u(k-1) yields uk, which is the same as v such that; so basically it is just this. Starting from u, you must be able to use the rules again and again. And successively yields strings till we get v.

So, we say u derives v, if we are able to repeatedly use the rules to get v. And like u and v could be general strings consisting of variables, terminals, combination of both everything. But, the language of the grammar is a set of strings that you can obtain that consists of only terminals.

So, once again the language of the grammar is a set of all w that can be generated. So, notice the w that can be generated using only the terminals, w from Σ^* such that starting from, you can get the w starting from the starting variable. So, you start from the starting variable s and are able to repeatedly apply rules till we get w. So languages, the set of all strings that are that only consist of terminals; and can be derived from the starting variable. So, you have seen enough examples already.

(Refer Slide Time: 09:42)



Why content-free?

$S \rightarrow aSb$

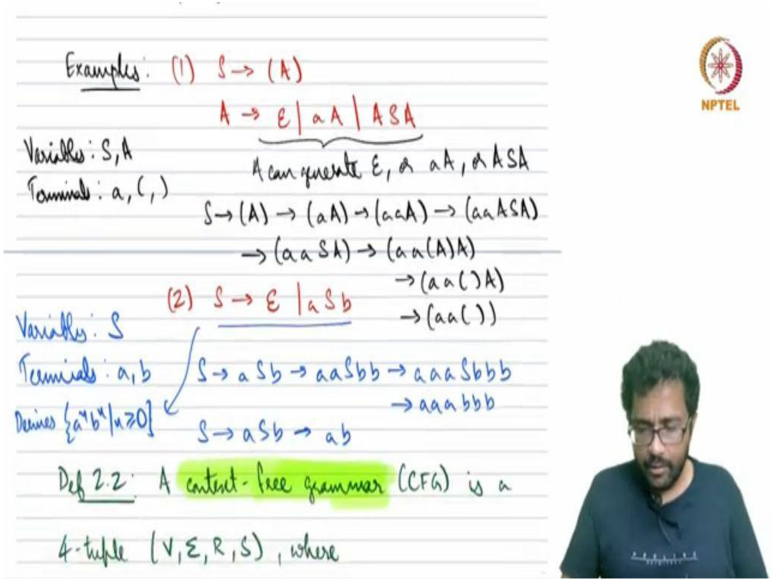
The rules and derivations are not based on the content of the variable. Rules are not of the type

$$aVb \rightarrow axb$$
$$bVa \rightarrow bya$$

$V \rightarrow x$

Rules do not depend on the content. There are content-sensitive languages, though we won't see them in this course.

[illegible]



Examples: (1) $S \to (A)$

$A \to \underbrace{\varepsilon \mid aA \mid ASA}$

Variables: S, A

Terminals: a, (,)

A can generate ε, aA, ASA

$S \to (A) \to (aA) \to (aaA) \to (aaASA)$

$\to (aaSA) \to (aa(A)A)$

$\to (aa()A)$

$\to (aa())$

(2) $S \to \varepsilon \mid aSb$

Variables: S

Terminals: a, b

Derives $\{a^n b^n \mid n \geq 0\}$

$S \to aSb \to aaSbb \to aaaSbbb$

$\to aaabbb$

$S \to aSb \to ab$

Def 2.2: A context-free grammar (CFG) is a

4-tuple (V, Σ, R, S), where

And just to quickly tell, why is it called context-free? The context-free, because the rules are defined like this. So, the rules are something like this; A gives this, A gives epsilon, A → a A, A → aSa, or S → aSb. So, this is all we are saying things like S → aSb; so, we are telling how to replace S.

We are not look and it does not matter what is to the left of S or what is to the right of S, where S is going to be replaced by this. So, think about the other situation where we have rules like this. Suppose, we have we have rules like this; so aVb, where V is some variable. So, now we see that V has a to its left and b to its right. And now if that is the case, we replace it V by x; so now we get a, x, b.

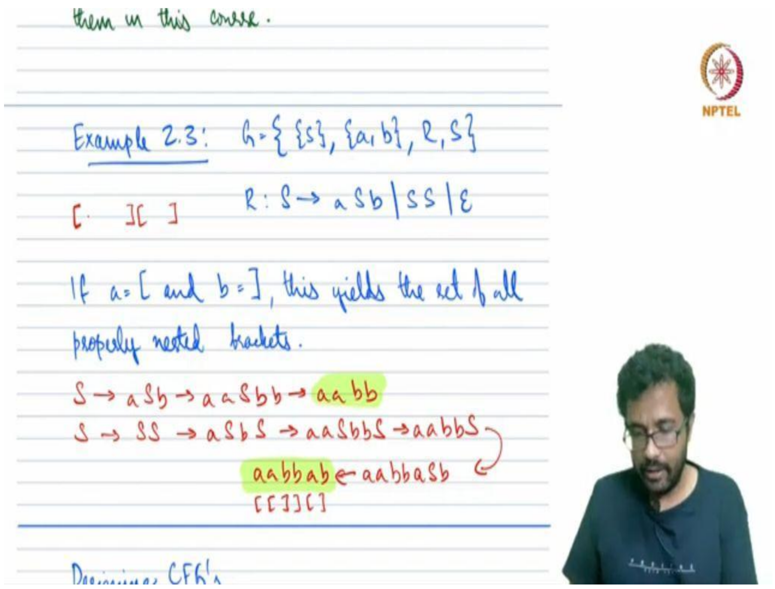
However, we had V had b to its left and a to its right; suppose it was of the form bVa. Then, you replace v by y. So, here what I have written here is an example of rules where we look at what is there to the left and right; and then tell what to replace it with. Meaning I am looking at the context in which the variable is surrounded, the context in which the variable is situated, context in which the variable is seen to determine which rule applies to the variable. So, this is what you would call context sensitive languages or context sensitive rules, where I have to look at the variable and look at the context in which we saw them.

However, all the variables are and all the rules that we saw so far in the case of context-free languages or context-free grammars, do not look at the context. They just saying that you look you see a V, and then V → axy or something like that. Without really or V gives, V gives x without really bothering about what is V surrounded by.

So, that is why this is called context-free grammars. The rules and the productions are independent of the context in which the variable is found; hence, it is called context-free grammar. So, rules do not depend on the context in this case. There are cases or there are we can form grammars and languages where we bother about the context.

So, this is called context sensitive grammars or context sensitive languages, et-cetera. However, we are not going to be covering them in this course; so, in this course we will only be looking at context-free languages. So, this is the reason for the nomenclature or the name context-free grammar.

(Refer Slide Time: 12:46)


them in this course.

Example 2.3: $G = \{ \{S\}, \{a, b\}, R, S \}$

[ ][ ] $R: S \to aSb \mid SS \mid \varepsilon$

If a = [and b =], this yields the set of all properly nested brackets.

$S \to aSb \to aaSbb \to aabb$

$S \to SS \to aSbS \to aaSbbS \to aabbS \to aabbaSb \to aabbab$

[[]][]

Designing CFG's

Now, let us see, so I said earlier that we will see why properly nested parentheses or properly nested brackets are context-free or are context-free language. So let us see why. So, this is the grammar; the grammar has only one variable S and two terminals small a and small b. So, I am just using a and b instead of open bracket and close bracket, just because of the ease of writing.

And there are three rules. In fact, R S consists of three rules, but I am just writing them like this in shorthand, a bit of shorthand, a, S, b or SS or empty string. So, let us see some situations. So, S gives a, S, b; again S g→ aSb. Let us say S → e , so you get aa, bb. Maybe S → a Sb; sorry I will use another starting rule. I will use S $\rightarrow$ SS, maybe now there are two SS. Let us deal with the first S; aSbS; first S again aa, S, bb, S, aa, bb, S. Maybe I am just going to come over here.

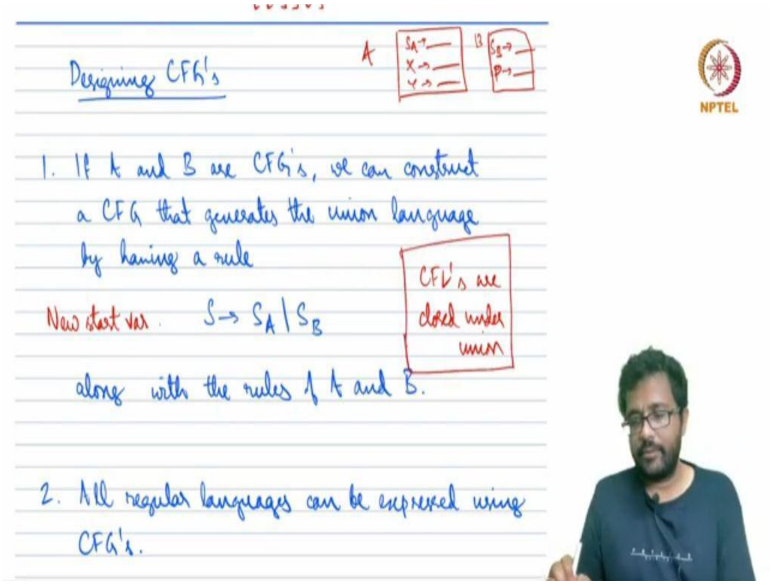
So, aa, bb, a, S, b, aa, bb, ab. So, here also I, the resulting string is just highlighting the resulting string aa, bb, ab; the first time we got aa, bb. So, you can see two strings; in both the cases, so if you replace a with open bracket, so you get; and b with close bracket, you get this. And you can see that this is indeed properly nested parenthesis or properly nested brackets.

And in fact you can see that any string that consists of properly nested brackets can be generated using this, so you come up with anything. So, the vague the high level idea is a following. So, either the string is of this form; so you, the starting has to be an open bracket, you cannot start with a closed bracket. Because, if the starting is with a close bracket, then you are not properly nested anyway. If the starting string, open bracket, if the partner of that open bracket is the very last symbol and the rest of the string is something in between then it is like we are using the rule a, S, b; S $\rightarrow$ aSb.

However, if the partner close bracket was in the middle somewhere, and then there is again an opening bracket and again a closing bracket; then, it is like we use the rule SS to begin with. So, this is a high level idea. But, it can indeed be verified that this generates all the strings that correspond to the properly nested brackets; so, it is a very simple context-free grammar. Only one variable S, only two symbols a and b. And three rules, S gives a, S, b; S gives SS and S gives empty string.

So, this yet another, so we already saw 0^n1^n which is not a regular language, but it is a context-free language. And now we have the set of all properly nested parentheses which is not a regular language, but is a context-free language.

(Refer Slide Time: 17:15)


Designing CFG's

A $S_A \to$ —, $X \to$ —, $Y \to$ — B $S_B \to$ —, $P \to$ —

1. If A and B are CFG's, we can construct a CFG that generates the union language by having a rule

New start var. $S \to S_A \mid S_B$

along with the rules of A and B.

CFL's are closed under union

2. All regular languages can be expressed using CFG's.

Now, let us see some simple observations. We already saw some observations, but now let us see two more. Suppose A and B are two context-free grammars. So, I am saying A and B be the two context-free grammars. Let and meaning we have a separate set of rules. So, A generates some language and B generates some language.

Now, can we design a grammar that generates the union of whatever A generates, and whatever B generates? So, we know that union is a regular operation and regular languages are closed under union. Is it true for context-free languages? So, you have some grammar A with which has some rules, let say some S_A let us be the starting thing, some other thing X gives something, Y give something et-cetera. And similarly you have S_B be the starting variable, let us say some p something.

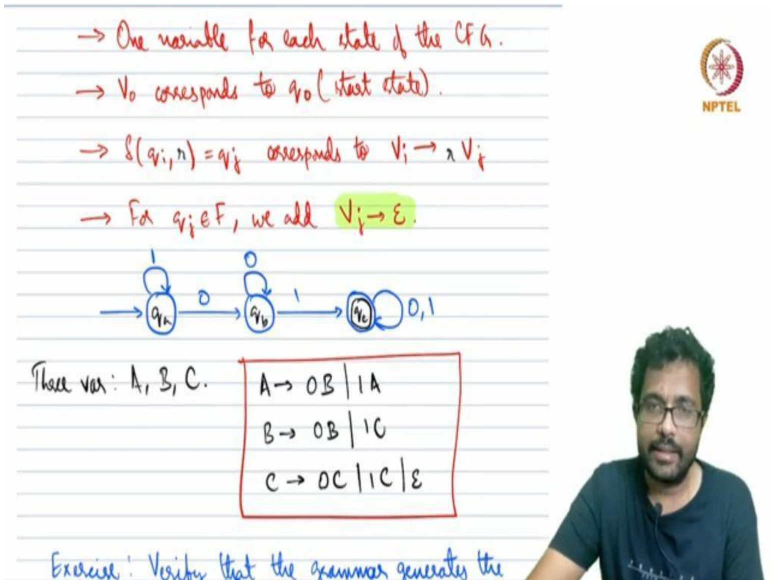
Now, can we construct a grammar that generates a union set of strings? The answer is yes; it is very easy. You create a new; you combine the rules of both A and B. You create a new starting variable, which you call S; so S is called a new starting variable. This S, sorry S is a new starting variable. And this S produces either S_A or S_B; so these two rules, S yields S_A and S yields S_B; and then we have all the rules of A and all the rules of B.

So, now either S can yield S_A; and then when S_A can yield any string that is generated by A; or S can yield SB and this B can yield any string generated by B. So, this is how we can get any string generated by A and B which mean the union language; so context-free languages are also closed under union. That is the learning here. CFLs are closed under union.

Now, we already said that this is a bit more than regular languages, but we also saw examples where there were languages that were not regular, but are CFGs. But, are every regular language, can we express every regular language using context-free grammar? Meaning our regular languages a subclass of context-free languages or a subset of context-free languages; the answer is yes. Every regular language can be expressed using a CFG; so, let us see how.

So, that is the way to do it if you have a DFA; you start with the DFA, if a language is regular there is a DFA. You consider you start with the DFA; you have one variable for each state of the CFG. And then corresponding to each transition you have rules and that gives us the grammar.

(Refer Slide Time: 20:42)



→ One variable for each state of the CFG.

→ V_0 corresponds to q_0 (start state).

→ $\delta(q_i, x) = q_j$ corresponds to $V_i \rightarrow x V_j$

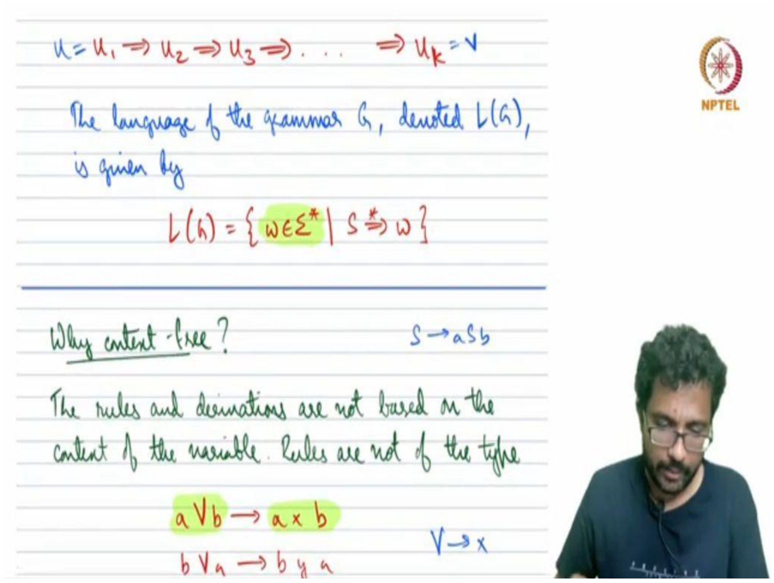
→ For $q_j \in F$, we add $V_j \rightarrow \varepsilon$.

$$\begin{tikzcd} \arrow[r] & q_a \arrow[loop above, "1"] \arrow[r, "0"] & q_b \arrow[loop above, "0"] \arrow[r, "1"] & \boxed{q_c} \arrow[loop right, "{0,1}"] \end{tikzcd}$$

Three var: A, B, C.

$$\boxed{\begin{array}{l} A \rightarrow 0B \mid 1A \\ B \rightarrow 0B \mid 1C \\ C \rightarrow 0C \mid 1C \mid \varepsilon \end{array}}$$

Exercise: Verify that the grammar generates the



$u = u_1 \Rightarrow u_2 \Rightarrow u_3 \Rightarrow \ldots \Rightarrow u_k = v$

The language of the grammar G, denoted L(G), is given by

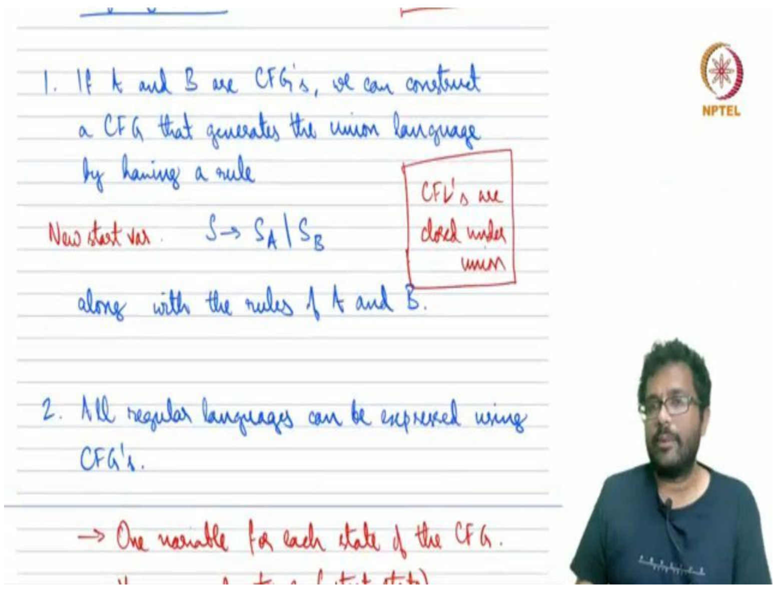
$$L(G) = \{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \}$$

Why context-free? $S \to aSb$

The rules and derivations are not based on the content of the variable. Rules are not of the type

$aVb \to axb$ $V \to x$

$bVa \to bya$



1. If A and B are CFG's, we can construct a CFG that generates the union language by having a rule

New start var. $S \to S_A \mid S_B$

along with the rules of A and B.

CFL's are closed under union

2. All regular languages can be expressed using CFG's.

→ One variable for each state of the CFG.

Let us see in this case, so this is a DFA. In fact, the language recognizes all the strings that contain 0, 1 I believe. So, let us see what are the things that we need three. So, there are three states here: small a, small b, small c. So correspondingly, we will have three variables; in this case three variables capital A, capital B, and capital C.

Now, let us see what are the rules; so capital A corresponds to the state small a. So, if there is a rule delta qi, a, so there is a small issue here. Because I am using a for the alphabet in the case of a DFA; and here a is called a state. Maybe I will do a small change; so, I will call it qa, qb and qc. So, qi comma a equal to q_j; maybe I will make it further simplified, so maybe I will say r or something. So, qi, r equal to q_j gives us V_j, r to V_j.

So, the rules here are, so A gives, when you upon seeing a 0, you go to B. So, the corresponding rule is 0 and B. And upon seeing a 1, you go to itself; so you have 1 A. So, A yields 0 B; and A yields 1 A. And similarly B yields also 0 B, because upon seeing a B from 0, or upon seeing a 0 from B, you go to itself. Upon seeing a 1, you go to C, so 1 C and C upon seeing a 0, you go to C; and upon seeing a 1, you go to C. However, you notice that this is not a complete this is not a complete context-free grammar, because this will every on the left, on the hand side of every rule there is always a variable; so this is not going to help.

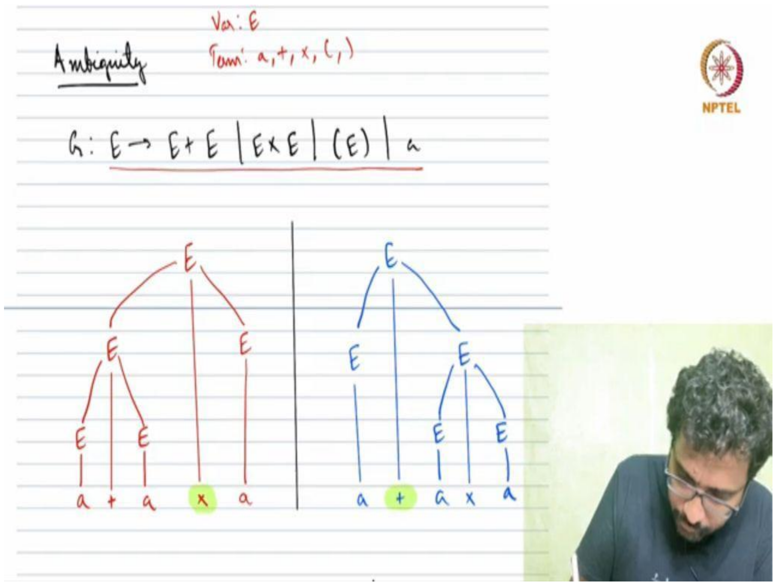
So, where are we going to get rid of variables? So there for that purpose, we have this final set of rule here. So, whenever there is a state which is an accepting state, we will add if q_j is an accepting state; we add Vj equal, Vj yields empty string. So, here qc is an accepting state; so we add c yields an empty string.

So, now you can try playing around and seeing what this context-free grammar yields. And this will yield the exact same set of strings that this DFA accepts nothing more, nothing less. So, either you can verify that this yields exactly the set of strings, which is the set of all strings that contain the substring 0, 1; so, this you can verify.

So, so far we have seen context-free grammar, we have seen examples, we have seen, we have seen the definition. We have seen why they are interesting; we have seen what is called the parse tree. We have seen the formal definition, we have seen what are the set of strings that is derived from the context-free grammar. We saw that context-free grammars, context-free languages are closed under union. And we saw this thing that all regular languages are expressed using context-free grammars.

So, whatever we did here is something general. So, given any DFA we can write a context-free grammar such that this context-free grammar generates the same language as a DFA. So, finally, there is one more small concept that I want to just quickly talk about before closing the lecture.

(Refer Slide Time: 25:58)

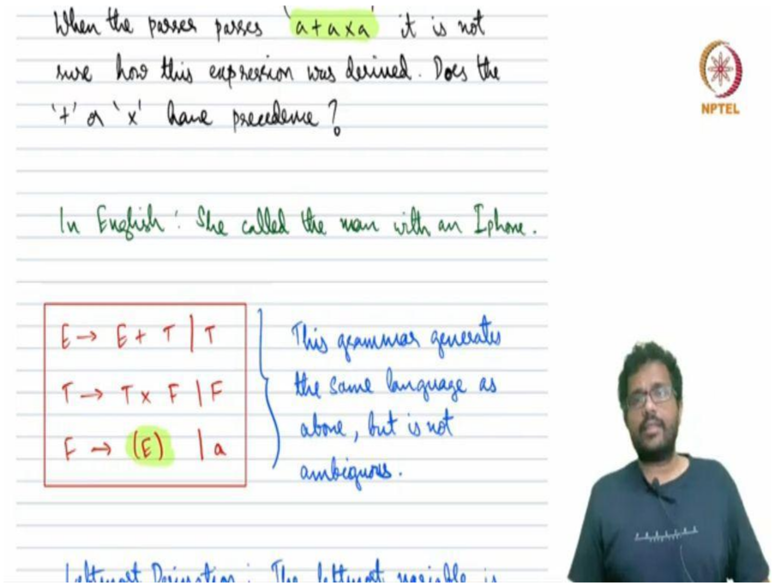


Var: E

Term: a, +, x, (,)

Ambiguity

G: $E \rightarrow E+E \mid E \times E \mid (E) \mid a$



When the parser parses 'a+a×a' it is not sure how this expression was derived. Does the '+' or 'x' have precedence?

In English: She called the man with an Iphone.

$$\boxed{\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T \times F \mid F \\ F \rightarrow (E) \mid a \end{array}}$$

This grammar generates the same language as above, but is not ambiguous.

Leftmost Derivation: The leftmost variable is

This is called ambiguity. So, ambiguity means like confusion or uncertainty, let us see. So, suppose, we have a grammar like this, so the grammar is there is only one variable. So, one variable E and terminal there are multiple terminals. There is a small a, there is a plus symbol, there is an x or multiplication symbol, there is an open bracket and there is a closed parenthesis; so, there are five terminals. And there are four rules; E gives E + E| E x E | (E)| a.

Now, let us try to see two derivations of this grammar. So, if you see the left hand side, this one, the one that I have drawn in red, so, E first yields this E x E, and the E in the right side E gives a;

and the E in the left side gives a plus a. So, here we get this thing a + a x a, a plus a multiplied by a.

Whereas, in the derivation the right side denoted by blue, the first rule is E gives E + E, where the E in the left side gives a, and E in the right side gives a multiplied by a. So, in both cases we get, we end up with the same string a plus a multiplied by a. However, the one in the red, the one in the left side, the multiplication happens after at the last; so, the multiplication happens at the top level. Whereas, in the one in the right side where in the blue color, the addition happens at the top after the multiplication.

So, both strings have been generated from the same grammar. But, both of them are the exact same string also, the same string; they are generated by the same grammar, but in two completely different ways. So, even if you look at these trees, they are structurally different. So, here, the x happens at the top level and here the plus happens in the top level.

So, if the x happens at the top level, it means that the X has precedence, the multiplication has precedence; sorry, the x happens after the addition. There is a plus happening at the top level means plus happens after the multiplication. So, the procedure really depends on which happens first. And this has to be actually included in the grammar. But, if we just see the string a plus a multiplied by a; so, think about a computer program, you are writing code.

Let us say you write 6 + 10 x 5, 6 plus 10 x 5. You are just writing it in the code, or x plus y multiplied by z. Now, when the compiler reads it and interprets it, how should it interpret it? Either the plus should happen after the multiplication or the multiplication should happen after the addition. It depends on how the grammar is defined. But, one thing is clear, it is not desirable that the same string x + y x z, both of them are both possible. If this is a string that you are reading, it should not have two different meanings; because that will give the compiler some confusion.

So, depending on what number system you are working with, perhaps multiplication takes precedence over addition or perhaps the opposite; but, it is not desirable that the situation is unclear. So, you want the situation to be quite clear as to whether addition takes precedence over multiplication, or the other way around. So, what is the issue here is that it is not clear which one takes precedence. Or in other words, it is not clear in which derivation was used to generate this or produce this string plus a multiplied by a. So, this is an undesirable situation; and this undesirable situation is called ambiguity.

So, ambiguity is by definition ambiguity means that the same string can be generated in two different structure in different ways, which is what is happening here. So, even in English sentences, sometimes this confusion can be there.

But, English is a natural language and sometimes these uncertainties can be there and somehow let us say for instance, this sentence, she called the man with an iPhone. So, is it that she used her own iPhone to call the man? But, she called the man with the iPhone. So, he she is using the iPhone to make the call to the man. Or, is it that with an iPhone is an adjective of the man so, there is there are several men and she is using, she is calling the man who has the iPhone. So, the same sentence has two different interpretations.

But, in common natural language conversations, there are contexts perhaps; or perhaps you, you can end up asking the other person usually the exact precision is not there. But, that is somehow usually clarified through interaction. Because usually common language you talk, you keep talking the conversation.

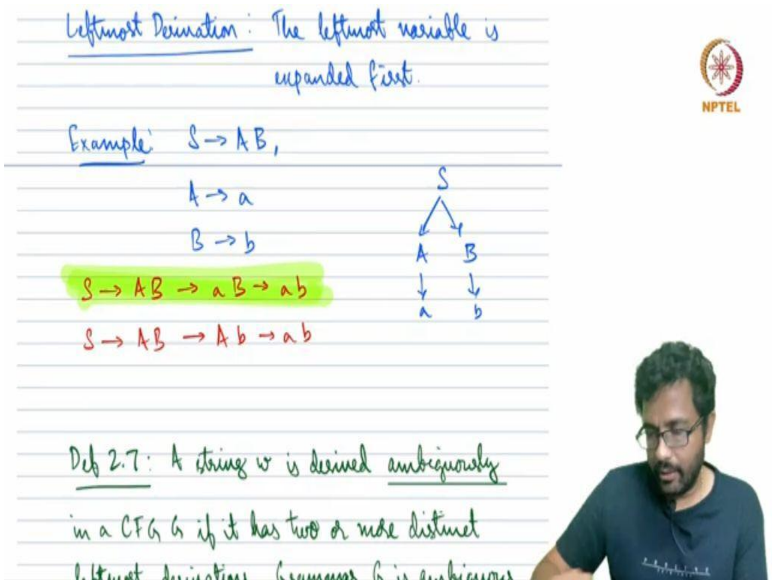
Whereas in a computer program, it is not like the computer program can always come back and ask such questions; it is like, you want precision, you want clarity, you do not want this confusion. So this is called ambiguity; this confusion is called ambiguity. And it is not a desirable thing to have in some cases, not in all cases. In some cases, we can actually take the grammar and modify it to get rid of the ambiguity.

So for instance, the grammar that is written here, the one in the top; I am talking about this underlined grammar; sorry, this grammar that I have underlined here, this generates some set of strings. The grammar that I have written here, this also generates the same set of strings. However, this one does not have ambiguity because it has a different set of variables.

So, here E can be replaced by E plus T, where T stands for a term and then T can be replaced with T multiplied by F, where F stands for a factor. So, a multiplicative factor will take will come after, the addition will come after the multiplication; so, multiplication clearly takes precedence here. And of course, there is always this provision for this parenthesis here, which we had here also; but we did not end up using it.

So, you can verify that this grammar generates the same language. And you can also verify that this particular kind of ambiguity does not get reflected here. So, this issue is called ambiguity and this is not desirable because in a computer program you do not want, you want the. You want to communicate with clarity as to what the operation, what is the addition or multiplication has to be that has to be performed.

(Refer Slide Time: 34:09)



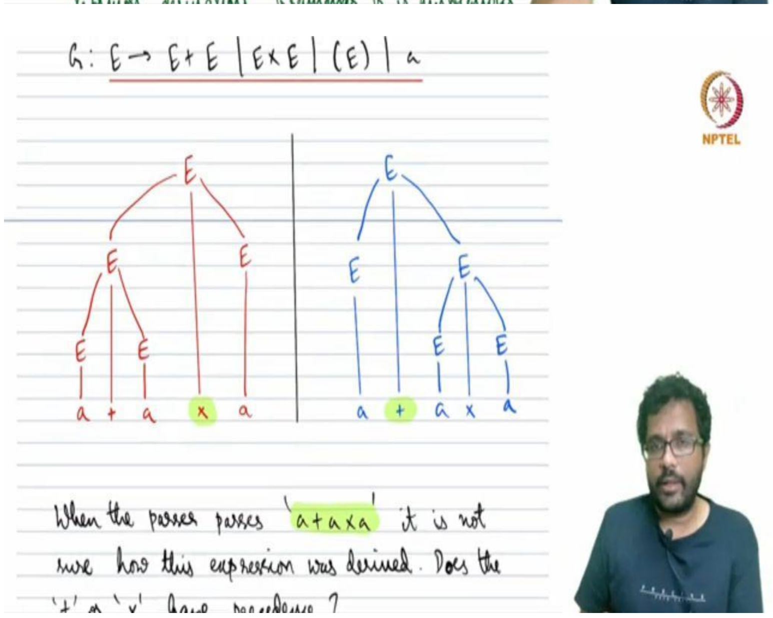
Leftmost Derivation: The leftmost variable is expanded first.

Example: $S \to AB$, $A \to a$, $B \to b$

$S \to AB \to aB \to ab$

$S \to AB \to Ab \to ab$

Def 2.7: A string w is derived ambiguously in a CFG G if it has two or more distinct leftmost derivations. Grammar G is ambiguous



$G: E \to E+E \mid E\times E \mid (E) \mid a$

When the parser parses '$a+a\times a$' it is not sure how this expression was derived. Does the '+' or '×' have precedence ?

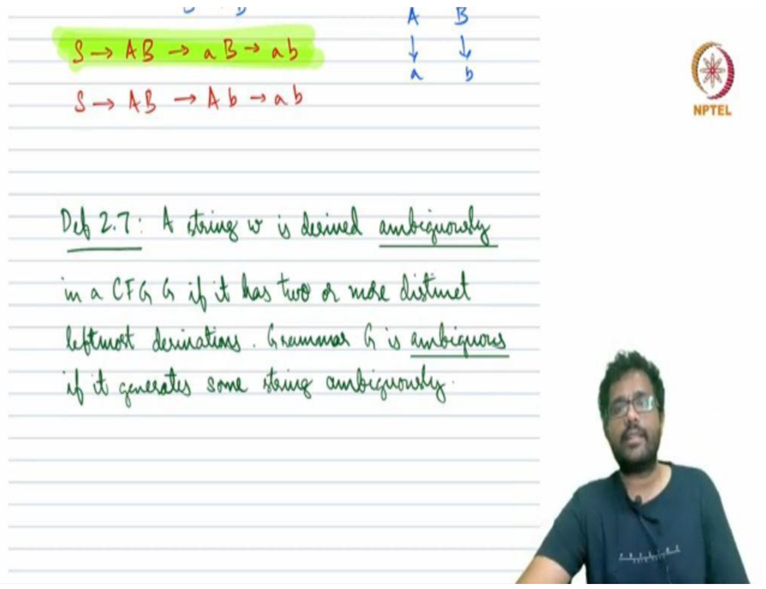
So, just a brief point about leftmost derivation; so, leftmost derivation means you always expand the leftmost variable. So, sometimes when you have a string of multiple variables, you expand the leftmost variable. So just quickly, so you have a very brief grammar here S gives AB, A gives small a, and B gives small b.

So, you could do this S gives AB, you replace the A with small a; then, you replace the B with small b. Or you could replace the B first and then the A; but, both of them are stuck. So, what I am saying is, there are two different ways to derive a small a, small b; but the only difference is that I am applying the rules in a different order. In other words, if I just try to draw the tree here, so it is like this; S gives capital A capital B, capital A gives small a, small b gives small b. This the same tree applies to both of these derivations.

So, suppose I had insisted that I always expand the leftmost variable first, then the second derivation, this derivation will be the only one that meets that condition. Because here we have A and B and sorry, not the second, sorry; only the first one will meet that condition. We have A and B, and we replace A first.

So, suppose I say that the leftmost variable always has to be replaced; then there is only one way to derive AB. However, here that was not the case. Even with the same leftmost derivation, there are two different ways to get it; so that is why we have two different trees also. So, a leftmost derivation corresponds to a certain tree; so here, both the derivations correspond to the same tree. So, I am just defining this leftmost derivation just to define formally what is ambiguity.

(Refer Slide Time: 36:26)

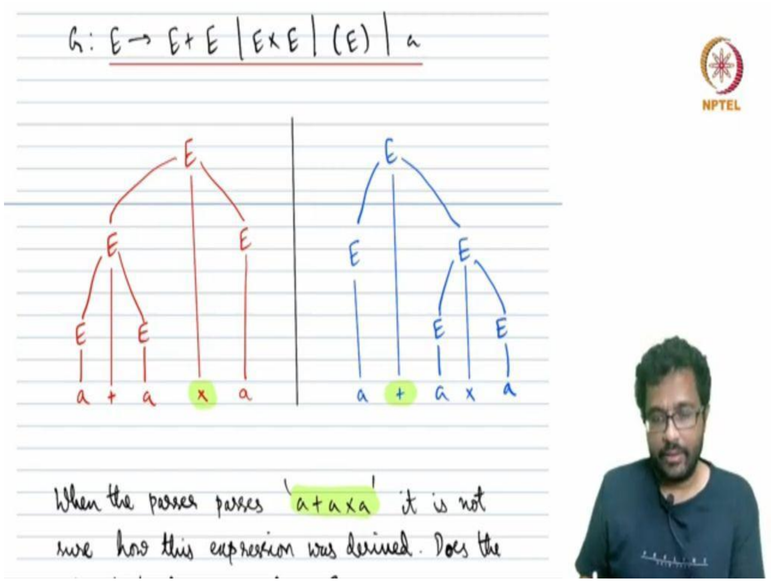


$S \to AB \to aB \to ab$

$S \to AB \to Ab \to ab$

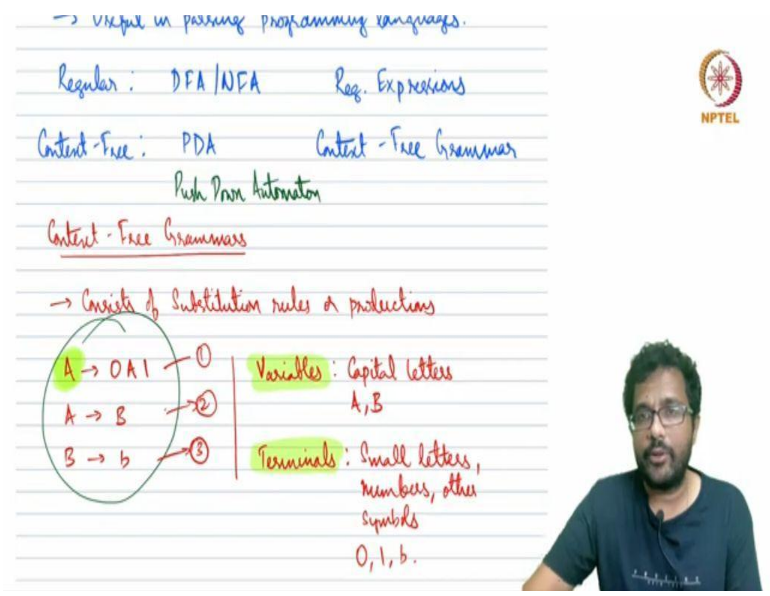
$$\begin{tikzcd} A \arrow[d] & B \arrow[d] \\ a & b \end{tikzcd}$$

Def 2.7: A string w is derived ambiguously in a CFG G if it has two or more distinct leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.



$G: E \rightarrow E+E \mid E\times E \mid (E) \mid a$

When the parser parses 'a+a×a' it is not sure how this expression was derived. Does the



→ Useful in parsing programming languages.

Regular : DFA/NFA Reg. Expressions

Context-Free : PDA Context-Free Grammar

Push Down Automaton

Context-Free Grammars

→ Consists of Substitution rules or productions

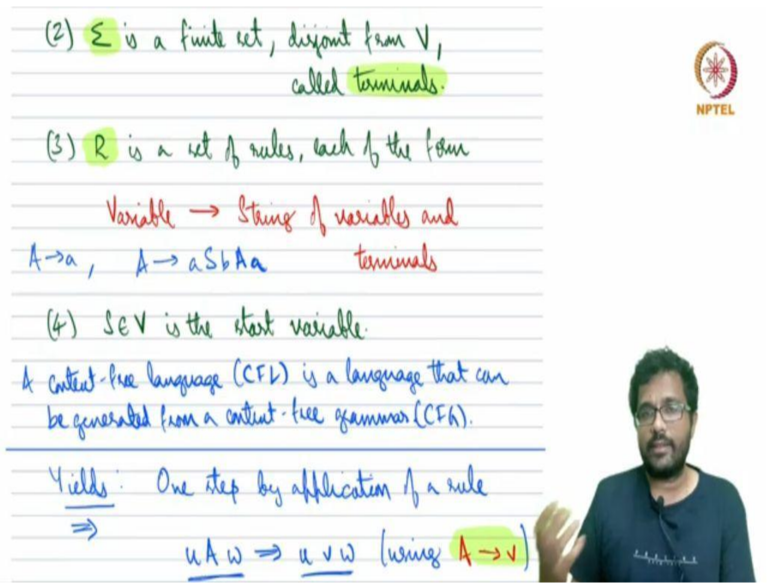
$A \rightarrow 0A1$ — ①

$A \rightarrow B$ — ②

$B \rightarrow b$ — ③

Variables : Capital letters A, B

Terminals : Small letters, numbers, other symbols 0, 1, b.



(2) Σ is a finite set, disjoint from V, called terminals.

(3) R is a set of rules, each of the form

Variable $\rightarrow$ String of variables and terminals

$A \rightarrow a$, $A \rightarrow aSbAa$

(4) $S \in V$ is the start variable.

A context-free language (CFL) is a language that can be generated from a context-free grammar (CFG).

Yields: One step by application of a rule

$\Rightarrow$

$\underline{uAw} \Rightarrow \underline{uvw}$ (using $A \rightarrow v$)

So, we say that a string is derived ambiguously, if it has two or more distinct leftmost derivations. So, this is an example, a plus a multiplied by a has two or at least two distinct leftmost derivations. So here, the E gets replaced by E x E, in the right side E gets replaced by E plus E, here E the left side E gets replaced by E multiplied by E first, here it is E plus E first.

Now, even if we do everything leftmost, they are structurally different; that is why the tree is different. So, a string is derived ambiguously in a grammar in a context-free grammar, if it has two or more distinct leftmost derivation and the grammar we say that grammar is ambiguous, if it has if it generates some string in an ambiguous manner. If there is some string that is generated ambiguously, we say that the grammar is ambiguous. So, that is the definition of ambiguity which is not a desirable trait.

So, I will quickly summarize this lecture, so, we introduced context-free languages. We said that there are two representations using pushdown automata and using context-free grammar. We are going to see context-free grammar, we defined context-free grammar, we saw some examples. We saw that we saw which are the strings that are generated from the context-free grammar; we saw which are the, what is the language that is generated from the context-free grammar. We saw why it is context-free.

We saw that regular languages are context-free, we saw that context-free languages are closed under union. And then we saw ambiguity and maybe formally saw definition as well. And with that, we will close week-3. In the next week, we will see more on context-free languages and context-free grammars. We will see other ways to represent them and other rules that pertain to them. Thank you.