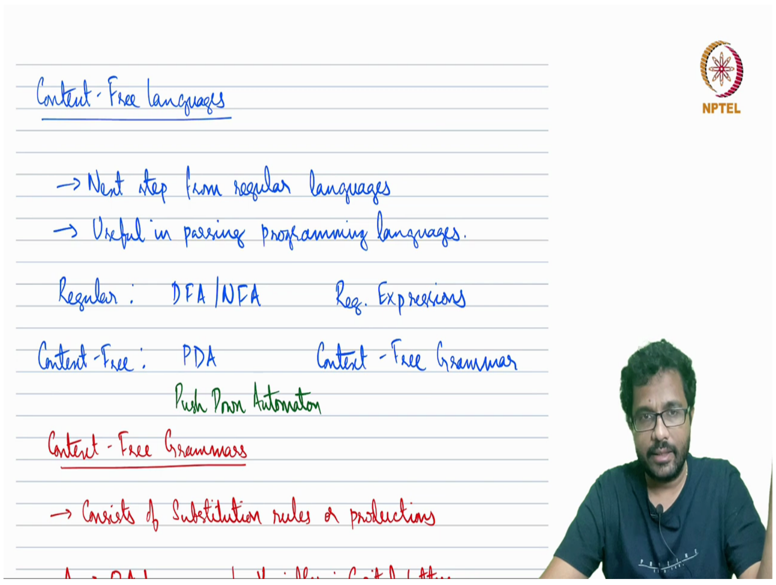
Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad
An Introduction to Context-Free Languages - Part 01

(Refer Slide Time: 0:16)



Context-Free languages

→ Next step from regular languages
→ 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

Hello and welcome to Lecture 16 of the course, Theory of Computation. So far in the course, we have seen regular languages. We have seen how they are represented. So we saw DFAs, NFAs, regular expressions. We saw pumping lemma which gave us a way to show how languages are not regular. We also saw Myhill-Nerode theorem.

So now, in this lecture, we are going to start on a new topic which corresponds to Chapter 2 in the book called Context-Free Languages. Actually, this is a; we can consider this as a next step from the regular languages. So this will be somewhat more general than regular languages. And one of the applications will be that it is useful in let us say compilers or parsing languages.

So for regular languages, we had 2 or 3 ways to represent them. So one was using DFAs or NFAs and one was using regular expressions. So DFAs and NFAs were basically a machine-based model. So we had an Automaton in either case and whereas regular expressions was like an expression that needs to be interpreted into a string. And you could do that in multiple ways.

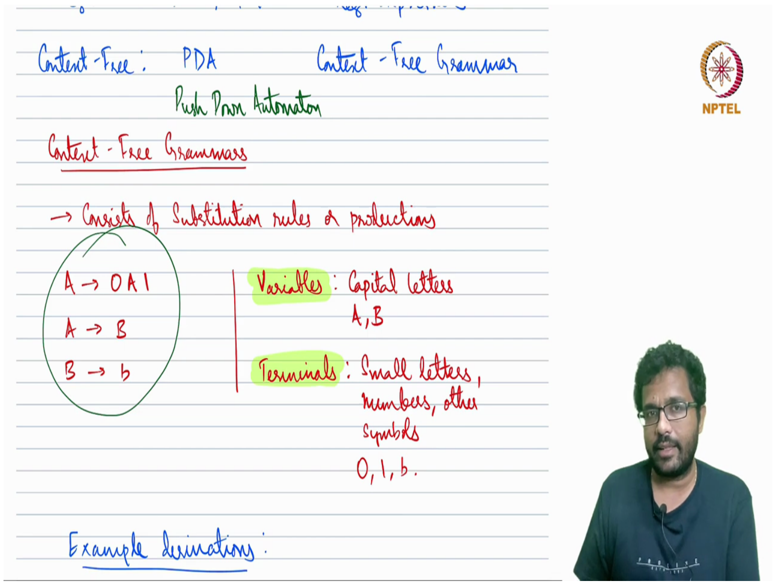
Similarly, in the case of context-free languages also we will see the same similar situation. We will have one machine-based model, which is called PDA. So PDA stands for Push Down

Automaton. And we will see a something like regular expressions called context-free grammar. So grammar-based representation.

So in regular languages, we had 2 machine-based representations whereas, in context-free languages, we only see one called Push Down Automaton. But in this in the case of context-free languages, we will first see the grammar-based representation called context-free grammar.

So the class of all languages that can be represented by context-free grammars is going to be called context-free languages. Later, we will see the other representation also. We will also show the equivalence but first, we will start with context-free grammars.

(Refer Slide Time: 2:34)



Context-Free: PDA Context-Free Grammar

Push Down Automaton

Context-Free Grammars

→ Consists of Substitution rules or productions

$A \to 0A1$

$A \to B$

$B \to b$

Variables: Capital letters A, B

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

Example derivations:

So as I said, already, it is more general than regular languages and have a more interesting applications. So we can do more things with context-free language. So let us start. So what a context-free grammars?

So basically, we have some variables and we have rules to interpret these variables. So here in the left side, we have these 3 rules. One is that A → 0A1. Then A gives then A → B, then B → b.

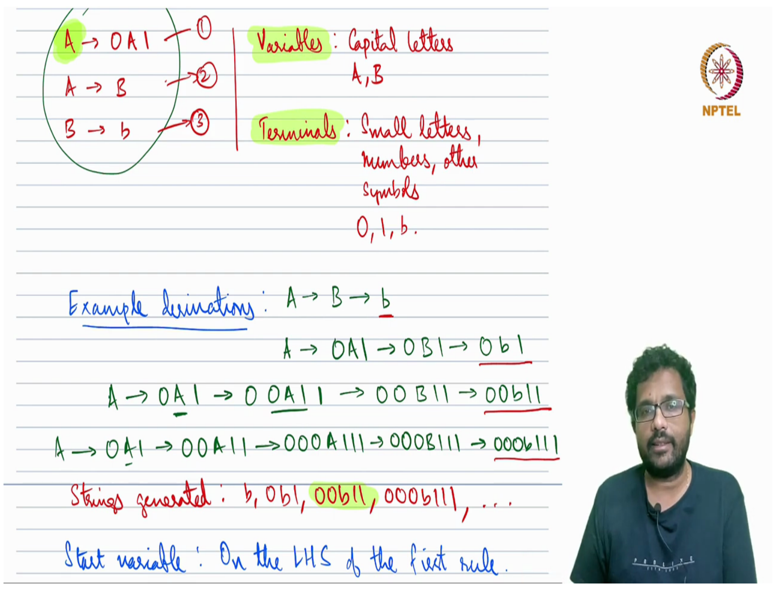
So we can think of the right arrow as a rule- of a substitution rule or a production rule. So A yields 0A1 or A yields B or B yields b. So what this means is that we can in a certain situation, we can replace A with 0A1 or we can replace A with capital B or we can replace B with capital B.

And typically what, there will be two kinds of objects that we will deal with. One is denoted with capital letters- A and B, called variables. So typically capital letters. So, in this example, it is just A and B and two is terminals.

So in this example, there are 3 terminals 0, 1 and b. So small letters numbers and maybe other symbols also sometimes. These denote terminals. So terminals means that you cannot further substitute them with other things whereas variables are variables. So there are ways to substitute them.

So here is A is a variable, capital A is variable. There are two different ways to substitute them. So you could replace A with 0A1 or you can replace A with B. Whereas, once you have small b, then you cannot replace them or you have 0 or 1 you cannot replace them with anything else. So a variable needs to be replaced and till you get a terminal.

(Refer Slide Time: 4:50)



$A \rightarrow 0A1$ ①
$A \rightarrow B$ ②
$B \rightarrow b$ ③

Variables: Capital letters
A, B

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

Example derivations: $A \rightarrow B \rightarrow \underline{b}$
$A \rightarrow 0A1 \rightarrow 0B1 \rightarrow \underline{0b1}$
$A \rightarrow 0\underline{A}1 \rightarrow 0\,0\underline{A}1\,1 \rightarrow 00B11 \rightarrow \underline{00b11}$
$A \rightarrow 0\underline{A}1 \rightarrow 00A11 \rightarrow 000A111 \rightarrow 000B111 \rightarrow \underline{000b111}$

Strings generated: b, 0b1, 00b11, 000b111, ...

Start variable: On the LHS of the first rule.

So let us see what all we get from this grammar. So let us try to see some examples. So one is that A yields 0A1 and now let us say or maybe I will use another one. So A also gives us B, and B gives us b. So now b cannot be, you cannot do anything with b because it is a terminal.

Another possibility is A yields 0A1. Now 0 and A, 0 and 1 are terminals but A you can again replace it with B and B again you can replace it small b. Now another way to do it is A gives us 0A1. Now we can again use the same rule to replace this A. So to replace the underlined A, we can again use 0A1. So this underlined A was replaced with 0A1.

And now let me say I have 00A11, so I am replacing it with a b, and then B with a b. I can do this any number of times. So A gives 0A1. Now, this A gives 0A1 again and this A again gives 0A1.

So, these are all strings that can be obtained. So notice the strings, the ending strings that we have. So the first case it was b, then it was 0b1, then 00b11, and then 000b111. So and now, you may be, you may kind of see the pattern here.

So strings generated are b, 0b1, 00b11, 000b111 and you can see the pattern here. And you can also convince yourself that these are pretty much all the strings that are generated look like this. So, b and then some 0s before b and some 1s after B where the number of 0s and number of 1s are the same. So these are the kinds of strings generated by this, this particular grammar.

And so now once we reach some string like this, let us say we reach 00b11. So now, we cannot do any replacement because 00b11, each one of them, the three terminals here 0, b, and 1. That 0 appears twice; 1 appears twice.

But these are terminals, meaning you cannot replace them with something else. You can only replace a variable. Variable in this case, there are two variables A and B, A and B, capital A and capital B can be replaced but small b and 0 and 1 cannot be replaced.

So the goal, even though I did not formally define it, I hope that the goal is now clear. To get a string, we need to start with A. So why A? I will just very soon tell you. So you look at the first; so I have written 3 rules here.

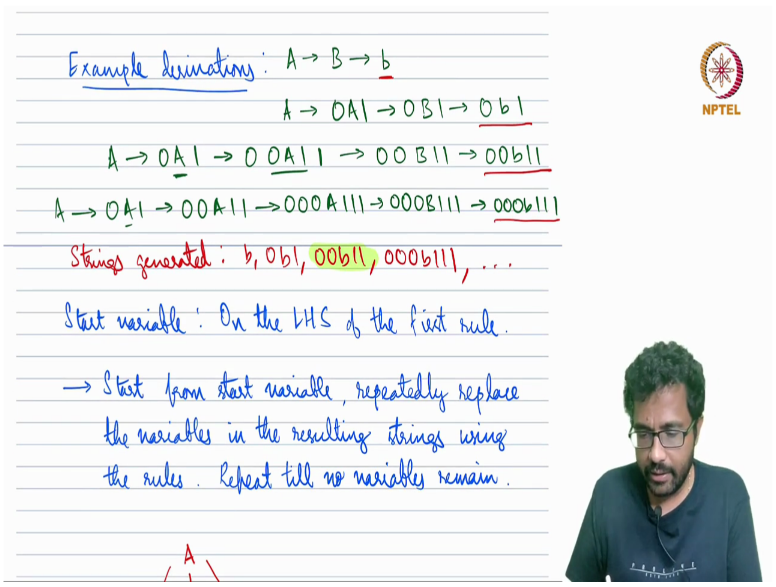
So there is one, there is 2. So this is rule number 1, this is rule number 2, and this is rule number 3. So you look at the rule, the first rule and see what is the variable that is replaced.

In the first rule, the variable that is replaced is A. The variable that is on the left side of the rule. So the rule is one variable and what to replace that variable with.

So the variable can be replaced by another variable. It could be replaced by one terminal. It could be replaced by two terminals or it could be replaced by two variables or a string of multiple variables and terminals, a combination, could be any of these.

So you look at the first rule and the variable on the left side of that rule, left-hand side of that rule that is the starting variable. Now you start with the starting variable and you get to some string that is entirely composed of terminals. And that is a string that is generated from the language. So in this case we saw multiple strings and you can get even more strings. In fact, there are infinite strings that can be generated.

(Refer Slide Time: 9:48)



Example derivations: $A \to B \to b$

$A \to 0A1 \to 0B1 \to 0b1$

$A \to 0A1 \to 00A11 \to 00B11 \to 00b11$

$A \to 0A1 \to 00A11 \to 000A111 \to 000B111 \to 000b111$

Strings generated: b, 0b1, 00b11, 000b111, ...

Start variable: On the LHS of the first rule.

→ Start from start variable, repeatedly replace the variables in the resulting strings using the rules. Repeat till no variables remain.

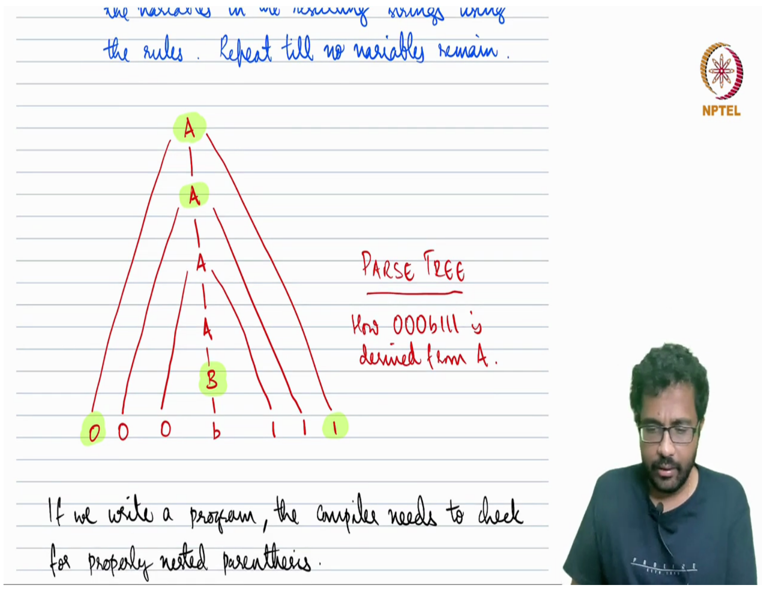
A

So again, I have written it in detail here. So the start variable is on the left-hand side of the first rule. So what you have to do is to start from the start variable and repeatedly replace the variables, you repeatedly replaced the variables in the resulting strings. We repeat using the rules, using the rules and you repeat like how many times you have to do this? You have to do this till there are no more variables.

So in this case, there are like we reach these strings, the underlying string. So there are and when we reached these, there are no more variables. So this is the goal and we will state it a bit more formally a bit further but this is how you get a string from a grammar.

So you start from the start variable and you repeatedly replace a variable with using a rule. So here we use we have 3 rules, replace A with 0A1, replace A with capital B, replace capital B with small b and you do that till you end up with a string that is consisting of only terminals or completely eliminate variables.

(Refer Slide Time: 11:05)



the variables in the resulting strings using
the rules. Repeat till no variables remain.

PARSE TREE

How 000b111 is derived from A.

If we write a program, the compiler needs to check for properly nested parenthesis.

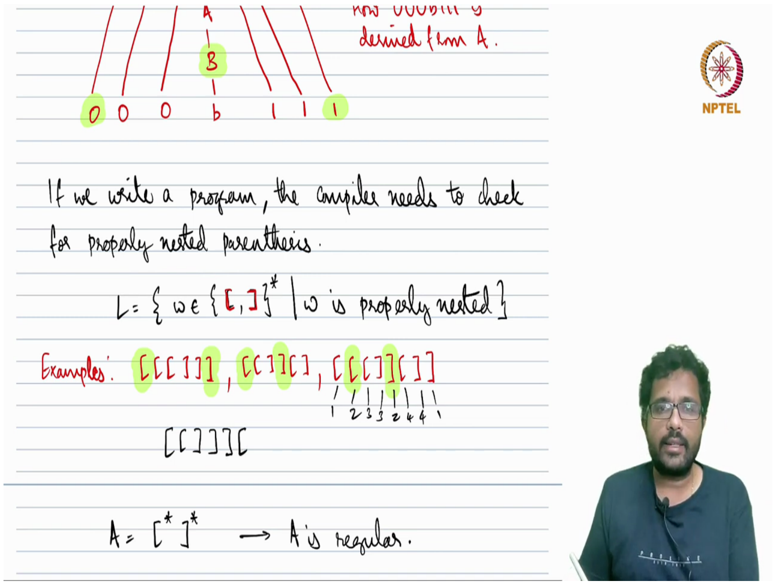
You can also draw a thing like this. This is called the parse tree. So this is a tree where we are seeing how 000b111 is derived from A. So we start with A. We started with this A and this a yielded 0, A, 1.

So this the topmost A yielded 0A1 and the second A again yielded 0A1. And the third A yet again yielded 0A1. That is how we got 000 to begin with and 111 at the end. But the fourth A just yielded B, capital B and the capital B just yielded a small b. And that is how we got the 000b111. So this is the same derivation but we are writing it in a tree form.

So tree form is probably easier to visualize instead of writing repeated derivations. And sometimes this is helpful because sometimes you have multiple variables. So in all these constructions that we had here, we just had one variable each time. Sometimes when you have multiple variables, this is easier to visualize.

And you have to replace one after the another in the case of derivation but in the case of a tree, you can just visualize them better. So this is what, this is the context-free language or this is the context-free grammar and how you derive strings from the context-free grammar.

(Refer Slide Time: 12:49)


A
B
0 0 0 b 1 1 1
derived from A.

If we write a program, the compiler needs to check for properly nested parenthesis.

$L = \{ w \in \{[,]\}^* \mid w \text{ is properly nested} \}$

Examples: [[[ ]]], [[ ]][ ], [[[ ]][ ]]
1 2 3 3 2 4 4 1

[[ ]]][

$A = [^*]^* \longrightarrow A$ is regular.

So let us try to see why this is interesting. Like what can you do in context-free grammar that for instance, we could not do in regular languages. So for instance, if you have written any kind of program or even something like using MATLAB or some kind of thing like that you need let us say when you have writing mathematical expressions, you need properly nested parentheses, parentheses or brackets, let us say.

So we meaning let us say so here I have written a language of brackets, brackets meaning the [. So set of all strings that are formed just using open square bracket ([ ) and close square bracket ( ]) such that it is properly nested.

So what are some examples? So I am just using square brackets instead of this parenthesis because this is easier to draw and visualize. So there is no curves there. So this is for instance, a string that is properly nested. Here is another one.

So in the first case, the starting bracket ended here, matched here, and then further there is two nested parentheses one inside the other. The second case is starting bracket closed here and within that there is one nested bracket and then outside that there is another nested bracket.

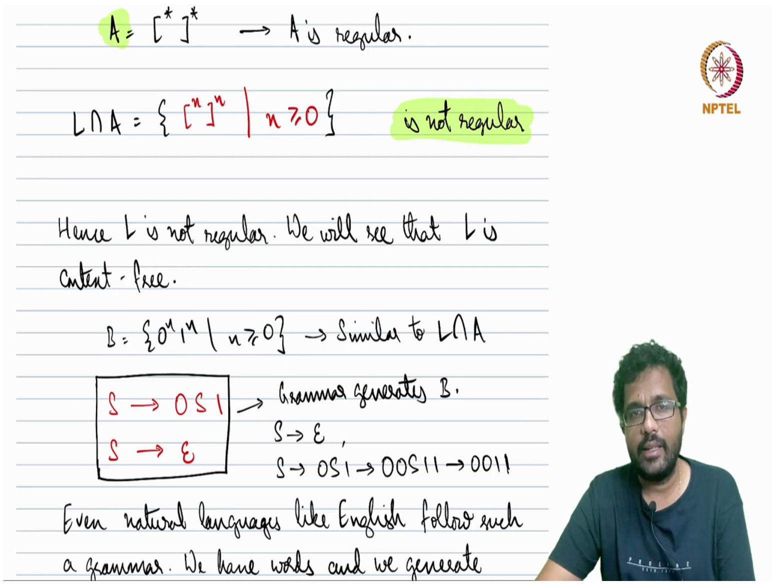
Here is another example. So I will just match. This matches with this. So you can see why they match. The maybe I will just note it down. This 1 matches with this 1; 2 matches with this 2; 3 matches with this 3; and then 4 four matches with this 4.

And you can see, what do I mean by properly nested. Properly nested means that the opening and closing should be in the right order. It should not be that so for instance this is not properly nested, because the closing comes.

So even this for instance let us say, this is not a properly nested language not a properly nested string because it does not even match. So for being properly nested, we need more than just matching, the number being matching because so here the number matches.

There are 3 open brackets, 3 closed brackets but this is not a properly nested one because you cannot, there is one closing off and then there is something opening. So that is the definition of properly nested.

(Refer Slide Time: 16:04)



$A = [^*]^* \longrightarrow$ A is regular.

$L \cap A = \{ [^n]^n \mid n \geq 0 \}$ is not regular

Hence L is not regular. We will see that L is context-free.

$B = \{0^n 1^n \mid n \geq 0\} \rightarrow$ Similar to $L \cap A$

$\boxed{\begin{matrix} S \rightarrow 0S1 \\ S \rightarrow \varepsilon \end{matrix}} \rightarrow$ Grammar generates B.

$S \rightarrow \varepsilon$,

$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 0011$

Even natural languages like English follow such a grammar. We have words and we generate

And this we know it is not a regular language. Why is it not a regular language? Let us see why. For instance, let us consider this language that is just open bracket star and close bracket star. So just two, it is just a alphabet of using two symbols that is the open bracket and the close bracket.

So the alphabet here is this, these two symbols and the language A that I am writing here, I am writing here. This is a regular expression over the alphabet and clearly, because it is a regular expression, it is a regular language. So some open brackets followed by some closed brackets.

So now, we had L here. And we have A here. What is L ∩ A? In other words, what is the set of all properly nested brackets which have the structure of A. So if you think about it, you will see that the only way to be properly nested within the structure of A is to have some number of opening brackets followed by the same number of closing brackets.

In other words, it has to be some number of opening brackets followed by the same number of closing brackets. So it is like open bracket n times followed by close bracket n times. This is L intersection A.

If L was regular, we know A is regular by the regular expression and hence if L was regular, L intersection A also would have been regular. But we know this L intersection A is not regular. We know this because we saw 0^n1^n when we studied regular languages and we know 0^n1^n for the same n is **not regular**.

So L ∩ A is; this is the same as 0^n1^n. It is just that instead of the symbol 0 we have open bracket and instead of the symbol 1 we have closed bracket.

So L intersection A is not regular. So this is clearly not regular. Hence, L is not regular. If L was regular, then L intersection A also would have been regular. But we can show that L is context-free. For instance, we will see a, we will soon see a context-free grammar that generates all the bracket pairs which are properly nested.

So, just quickly, I wanted to show L intersection A, similar to L ∩ A, a language. So let us say B. B is just 0^n1^n. It is the same as L ∩ A but instead of 0, instead of open bracket we have 0 and instead of closed bracket we have 1.

This the following the grammar in the box, this works. This generates, this grammar generates B. So you can see why it generates B for instance. So you start with S because that is the only variable here.

And then you can reuse, for instance, you can there are two things that you can do. You can do you can replace S with empty string in which case you have a string or you can replace S with let say 0S1 maybe this S again with 0S1 and then maybe with empty string.

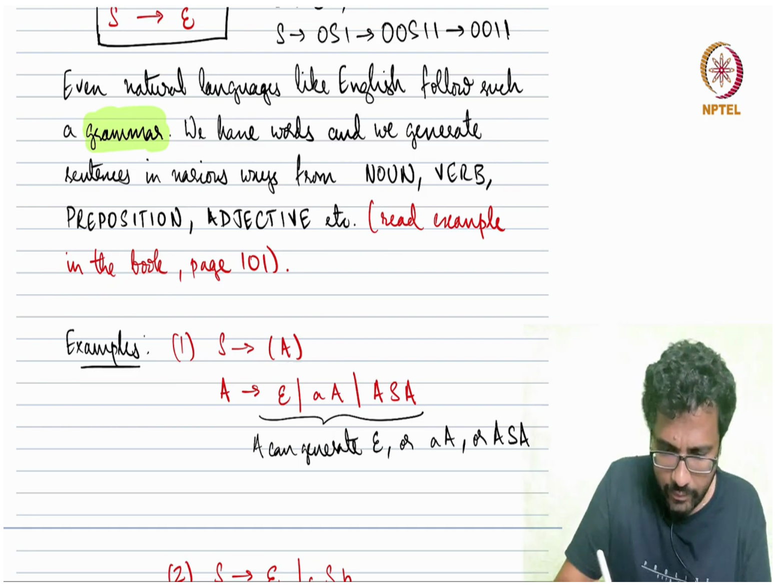
So you can see the strings that you generate are of the form 0^n1^n. So this B is context-free whereas meaning this is a context-free language meaning there is a context-free grammar that generates this language whereas we know it is not regular. So already we have seen an example, this is an example of a language that is not regular but is a context-free language.

And this is not the first time you are hearing the word grammar. So the word grammar you might have heard by learning languages in school English or any other like Hindi or any other local language that you might have heard or learned.

So we have grammar. So basically what is grammar? Grammar are just, grammar is just a bunch of rules as to how you structure your sentences, how you put together words to form meanings.

There are certain rules of what, what should go with what and how they should connect together etcetera, how to form sentences. The grammar in English is not the same as the grammar in the Indian language. So every language has its own specific set of rules and that is the grammar.

(Refer Slide Time: 21:33)



$\boxed{S \to \varepsilon}$ $\quad S \to 0S1 \to 00S11 \to 0011$

Even natural languages like English follow such a grammar. We have words and we generate sentences in various ways from NOUN, VERB, PREPOSITION, ADJECTIVE etc. (read example in the book, page 101).

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

$$A \to \underbrace{\varepsilon \mid aA \mid ASA}_{\text{A can generate } \varepsilon, \text{ or } aA, \text{ or } ASA}$$

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

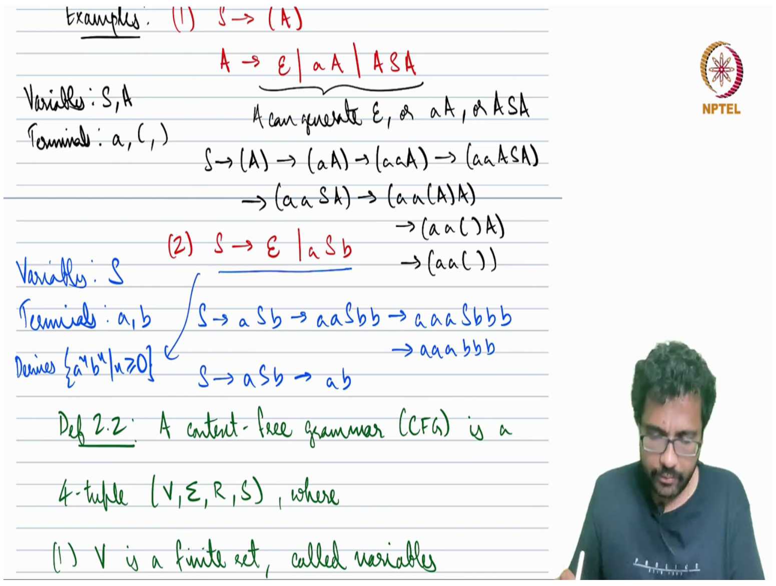
So even for natural languages such as English, we have the grammar. So in fact there is in the book there is an example of how you construct sentences, simple sentences using things like noun, verb, preposition, adjective, etcetera. I think in my copy of the book, it is in page 101. It is in the, it is towards the beginning of the Chapter 2. Please have a look at this. So I am not recreating, rewriting same example of how you can construct a sentence using these things like noun, verb, adjective, etcetera.

Now, let us see some more examples. So here we have context-free grammar. So let me just explain. So there are two there are two variables. One is that S, the other one is A and in the second row, second line, we have, we see these vertical lines. So I am talking about, I am talking about these vertical lines.

So this means that A can generate, this means that A can generate, this is actually 3 rules. One is that A can generate epsilon or it can generate small a capital A, or it can generate capital A S capital A.

So this is 3 rules. We are just, these **vertical lines denote OR**. All it is saying is that it is there are 3 rules. We are just writing them together. A can generate this or that or that. The first one, the first line says S generates open bracket, open parenthesis A closed parenthesis.

(Refer Slide Time: 23:27)



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

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

Variables: S, A

Terminals: a, (,)

A can generate ε, or aA, or ASA

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

$\rightarrow (aaSA) \rightarrow (aa(A)A)$

$\rightarrow (aa()A)$

$\rightarrow (aa())$

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

Variables: S

Terminals: a, b

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

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb$

$\rightarrow aaabbb$

$S \rightarrow aSb \rightarrow ab$

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

So what is the, what are the variables here? So the variables here are S and capital A. And terminals here are small a, that is a terminal. Empty string is an empty string. So that is, that is not a terminal. Then open parenthesis and close parenthesis. There are 3 terminals here, because they are also terminals.

So let us see some strings. So S generates, this is the first. The first rule has to be like this. Now A, let us see what it can do. Let us say A generates small a, maybe another small a and maybe let us say A generates ASA and maybe the first A generates an empty string, let us say and let us say S generates a, a, this.

Now let us say, I think we are running out of space. So I will just say both of the As, let them generate empty strings. We will do step by steps still. So this is a string that is generated. So I am just successively, every time I am replacing one of the variables here with, with using a rule with whatever is coming out of that. So I got this string open parenthesis aa open parenthesis again, close parenthesis, close parenthesis.

So this just one example of a language and here let us say this. Maybe I will just erase it and use a different colour for this. So here variables are there is only one variable S and terminals are small a, small b. Empty string is just empty string. It is not a terminal.

So here let us see S gives aSb. Maybe once more aSb and maybe once more aSb and maybe then we replace it with an empty string. So you have aaabbb. Another possibility is a Sb and then ab. Like this, we use the rule S equals the empty string. So here we can see this gives you a^n B^n.

So you can check that this gives you a^n B^n. So derive a power n and B power n, n greater than 0. So this you can check. I am sorry, I am bit cramped here for space but you can see what I am doing here. So hopefully by now, it is clear how you can play around with the context-free grammar and the kinds of strings that you can generate.