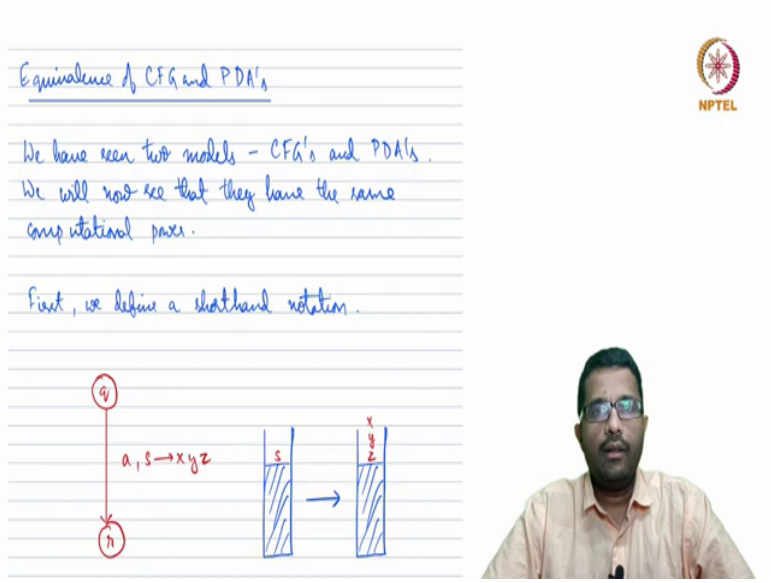
Theory of Computation
Professor. Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Equivalence of Context Free Grammars and Pushdown Automata - Part 01

(Refer Slide Time: 0:16)



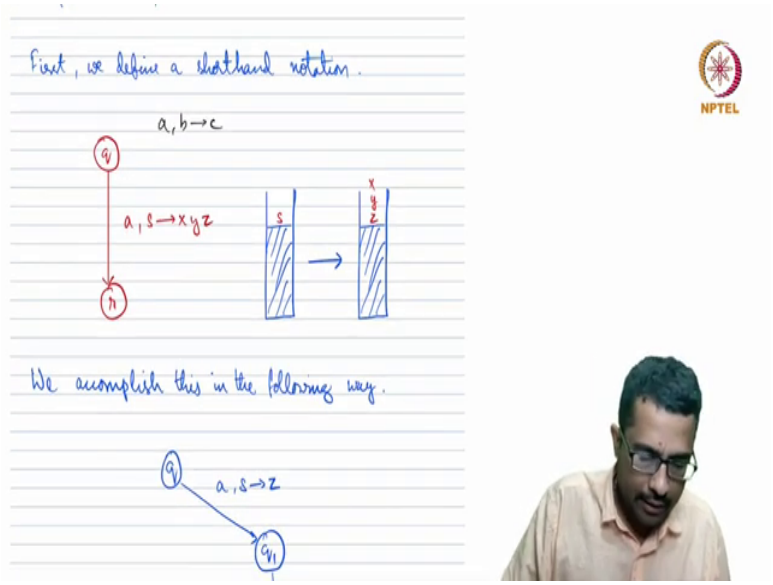
Hello and welcome to lecture 22 of the course Theory of Computation. In the past weeks we saw context free grammars and we saw that the languages that can be generated from context free grammars are called context free languages. And in the last couple of lectures, we had also seen another model of computation called pushdown automata and during that time I had perhaps remarked that these two models are going to be equivalent. And now we are going to see the proof of their equivalence.

So, one is a grammar based model, where you have expressions which can be interpreted in, where each variables are there and the variables can be replaced by bigger expressions and so on, that is the grammar based interpreter, the context free grammars. And pushdown automata is a machine model, so you have a machine that has rules for reading a string and then accepting or rejecting a string. So, at least superficially these two models look entirely different. One is a grammar based thing and one is a machine based thing.

And so, in that sense it is kind of interesting that they have the same computational power. But we had seen the similar phenomenon in regular languages as well, where we had a regular expressions which was like an interpretation based model where you have strings and rules for generating strings and we had DFA and NFA, in fact, two machine models which are able to read and process strings.

And we saw that all three of them have the same computational power. So, we had seen this sort of thing in regular languages. Now the same thing is repeating for context free languages. So, before we go to the proof of the equivalence, we will just briefly define a small shorthand notation.

(Refer Slide Time: 2:17)



So, suppose you, so far our notation was something like this, a,b -> c, when doing the transition between two states, implied that you read a from the input, you read b from the stack, meaning you pop that from the stack and you write c into the stack, so where a, b and c are all symbols; a is a symbol from the input alphabet, b is a symbol from the stack alphabet and c is a symbol from the stack alphabet as well, where a, b, c all could also be empty string. So, each one of them is a symbol of the respective alphabet or could be empty string.

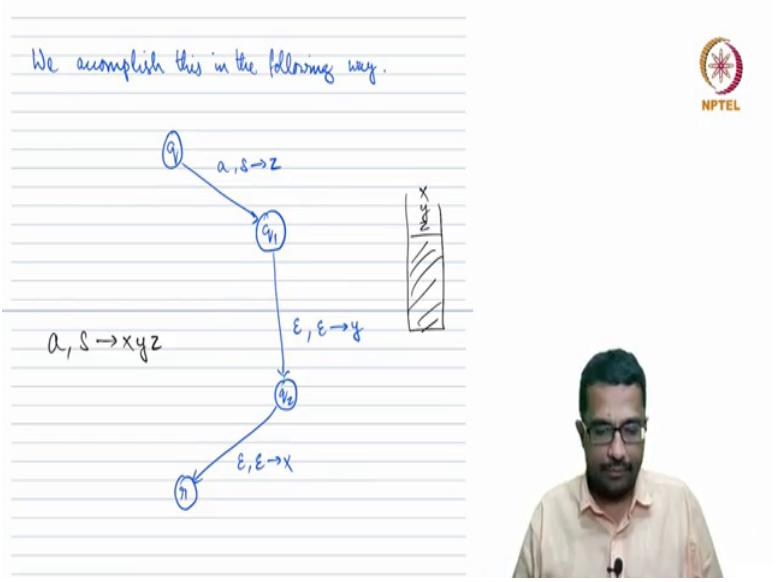
So, a means, so here a means, you read a from the input, b from the stack and you write c onto the stack. So, now we are going to see a slightly different type of notation where you say things like a,s -> x y z. So, what does this mean? This means that you, it is the same except that you have x y z here, where x y z is not a single symbol. So, you read a from the input that is provided to the machine, s from the stack, s is again a symbol or empty, and you write x y z into the stack, so x y z is a string consisting of the stack alphabet.

And so basically you want to say that instead of writing a single symbol you are writing a string, so this is what happens. So, you have some stack situation like this, so you have a stack and on the top of the stack, so you have something in the stack already. This blue, this shaded area shows something is there, and the top you have a symbol s on the stack, so you

read s from the stack, which means you pop that s off, which is a single symbol, and then you should write x y z into the stack. So, you have to input three different symbols or x y z is just, 3 is just arbitrarily chosen, you may want to write 4 symbols or 5 symbols or 10 symbols. So, this is what we want.

The stack should change from the top being s to top being x y z with the rest part remaining the same. So, this is a shorthand notation. So, we are going to use this notation in the upcoming proof and we can easily accomplish this. So, the transition here was from q to r.

(Refer Slide Time: 4:52)



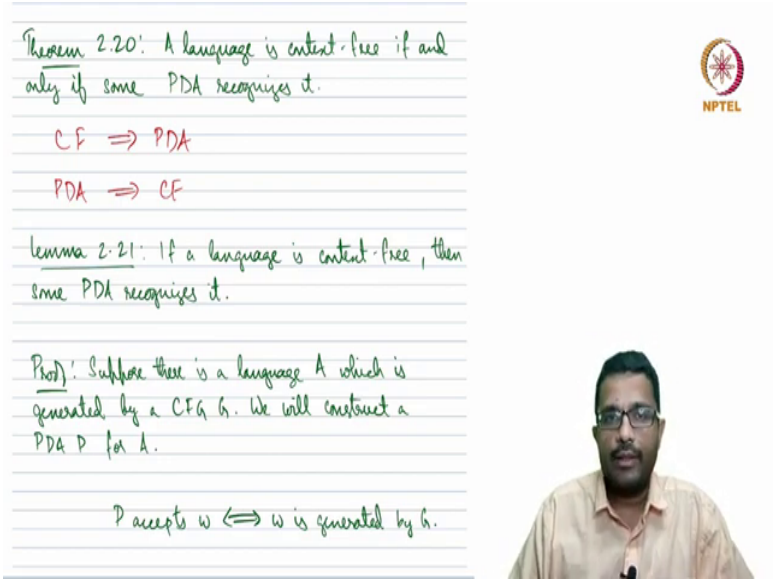
So, now we can accomplish this by introducing two new states q_1 and q_2, so q_1 and q_2, these are two new states, using these two states we can accomplish this. So, what we had to accomplish was a, s gives x y z. So, what we do is we add these two new states q_1 and q_2, so you make the transition from q to q_1, not q to r directly. So, you make the transition, where you read a from the input, so the input is already read, s from the top of the stack, so that is also red and then you write z into the stack.

So, when you write z into the stack, so the stack you put a z here, and then you make an epsilon transition from q_1 to q_2, where you read nothing from the input and you read nothing from the stack, but you write y onto the stack, so the stack here will add a y on top. And finally, you have a another epsilon transition, epsilon in terms of input as well as a stack, but you write x, so this is an error, so you write x. So, you write x onto the stack.

And once you do that, basically now you have replaced the top of the stack instead of s by x y z. So, from now on we will resort to this kind of shorthand notation a comma s arrow x y z,

so instead of a single symbol after the, on the right side of the arrow, we will have possibly a string and this can easily be accomplished by some extra states as we just saw. So, here we added two extra states. We may add we may require more extra states, so this is the first point.

(Refer Slide Time: 6:51)

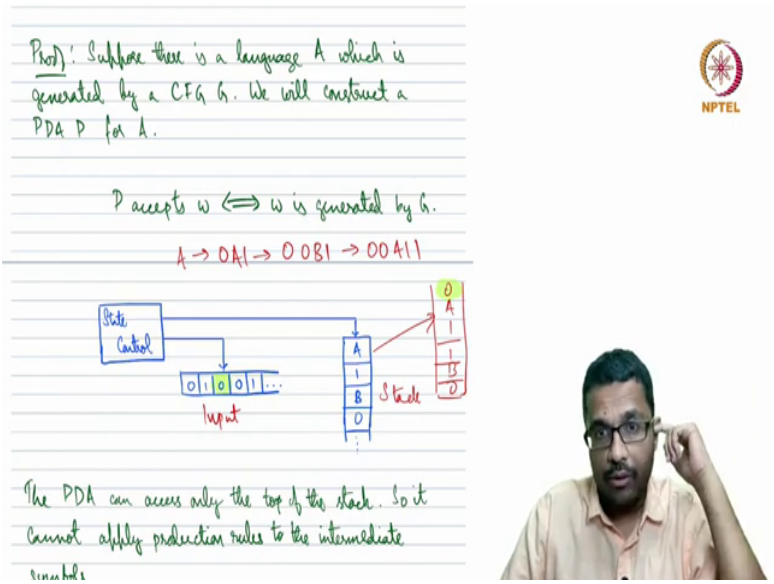


So now let us come to, let us state the theorem, the theorem says that - A language is context free if and only if some PDA recognizes it. So, there are two directions as always. The first direction is that a language is context free implies some PDA recognizes it, like context free implies PDA recognizes it and the other direction is that PDA recognizes it, implies that it is context free. So, this is the two directions that we need to show.

And what we do is we break down the proof into two directions, so the first direction, context free implies that there is a PDA that recognizes it is stated here in Lemma 2.21. And once again reminder these are the numbers from Michael Simpson, the edition that I have which is quite an old edition, but then I am just going to be referring to these numbers. So, we want to show that if a language is context free then there is a PDA that recognizes it.

So, context free so far means that there is a grammar for it. Context free grammar for it. So, from the grammar we will construct a PDA, the PDA will be equivalent and it will satisfy this property that whenever the grammar generates a string the PDA will also accept it and whenever the PDA accepts a string the grammar will be generating it, so this is equivalence. Whatever is generated by the grammar, exactly the same thing will be accepted by the PDA.

(Refer Slide Time: 8:24)



So, that is the goal. We have a grammar and we want to construct an equivalent PDA. So, let us try to see what the PDA can do. So, we have a stack, we have a state control and we have an input. So, the input is processed sequentially, you cannot go back or anything, you just read one symbol, one symbol, one after the other and the stack is this. So, let us for a moment assume that somehow, we are able to push the variables of the grammar, also the terminals of the grammar, so we had a grammar.

So, suppose let us say A here, and B here, are variables of the grammar; let us say 1 and 0 are terminals of the grammar. Suppose we are able to, the stack contains all of that. So, somehow, we want to be able to use the stack, to kind of use the stack or use this entire PDA structure, to emulate the derivation of a string from a grammar, so a string from a grammar you may derive it something like this, you may have something like a gives something 0 A 1 and then A may give 0 B, so the 1 remains.

And then I do not know B gives 1 a, let us say or A 1, something like this. So, now basically we have to keep doing this till we get rid of the variables in the string in the string on the right hand side and then we have a string that is generated from the grammar. So, now how do we, so this is the process of generating a string from a grammar. Now how do we kind of emulate that using this PDA setup?

So, one thing that kind of brings to mind is that suppose you have A on the top of the stack, you want to replace A with some rules. Let us say A gives 0 A 1. So, then you may want to replace A by like something like this, so 0 A 1, so A has to be replaced by this then followed

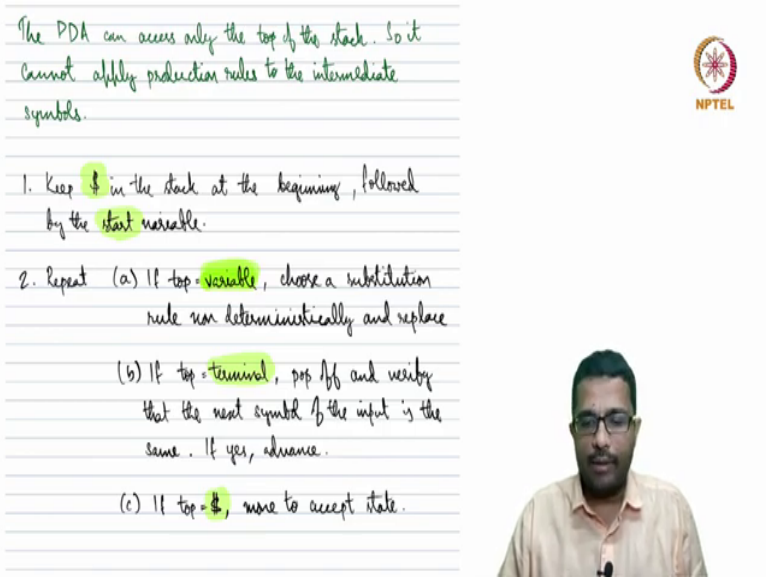
by whatever is already there 1 B 0 in the stack. Now when a is replaced by 0 A 1, now next is, then you see that the top here is 0, not A, not a variable but a terminal.

And then you see, when you have the top is a terminal, you cannot replace it because a terminal means you cannot replace it. So, then you check whether the next symbol of the input is the same as this. So, here if you see the next symbol of the input is a 0 and here also there is a 0, so you can kind of match them off and kind of proceed. So, you can move from the input symbol 0 here and you can pop this 0 off and these are the kind of things that we need to do.

So, there are two things that I mentioned here, one is if you have a variable at the top of the stack, you have to remove the variable and replace it by some rule which has, which replaces the variable. So, some rule of grammar. So, if you have A here, you look for a rule in the grammar where A is replaced by something. This is the first thing.

And if you have A terminal on the top of the stack, then you check whether the next symbol of the input is also the same, if they are the same you can move one step in the input, you can go to the next symbol, you just pop this, tops terminal from the stack. So, this is how, these are the two operations that we want to do. So, these are the two operations that we want to do and this will, basically we will be able to do exactly the same kind of things that a grammar does while generating a string. So, maybe let us just see one example or perhaps we will just describe it and then see the example.

(Refer Slide Time: 12:44)



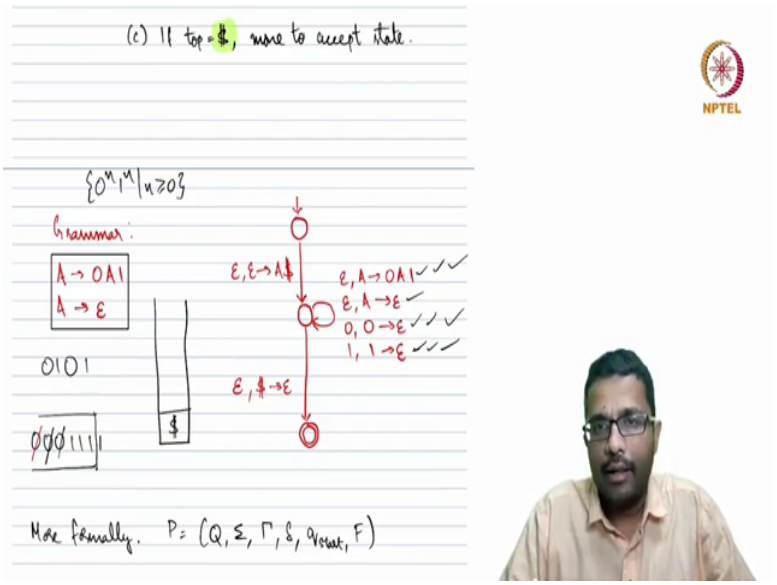
So, what we do is a following, so initially we keep a dollar symbol in the stack. So, this is like before, when we want to match up things, so we want to determine whether we have come to the end of the computation that is why we put the dollar at the beginning. So, again at the end we will ensure that there is a dollar, and that is when we go to the accepting state, so that is the purpose of this dollar.

And the other two cases are you just look at the top of the stack, if the top is a variable that is one case, and the top is a terminal that is another case. So, initially you put a $ followed by the start variable, so that is something I missed to say, so you put a dollar followed by the start variable. So, initially the first thing that you have to do is probably replay the start variable with some other variable, and then some other…

Start a variable $ using some computation derivation rule and then you just keep seeing what is there on the top. So, if the top is a variable, if the top is a variable, you replace it with some substitution rule, so it could be anything, there could be multiple rules, there could be a rule that says A replaced by 0 A 1; there could be a rule that says A replaced by C1. So, you could pick arbitrarily or not deterministically.

And if the top is a terminal, you just check whether the next symbol in the input, next symbol of the input, whether that is the same as the variable in the top of the stack. If they are the same, you move to the next step. If they are not the same you cannot take the transition. So, then when we do this, the only strings that will be accepted are strings that will be generated by the grammar.

(Refer Slide Time: 14:46)



So, let us just see an example. Here is a simple grammar, A gives 0 A 1 and A gives epsilon, so you may be able to verify that the language generated by this is actually 0 power n 1 power n. So, A is the only variable here. So, what do we do here? We first insert $ followed by the start variable, which is A here. So, initially the stack, so let us try a working example. Initially the stack has $, then A and then you come to the middle state.

There are three states only. And then there are four rules, the first one reads epsilon from the input and then replaces A. Even the second one reads epsilon from the input and replaces A. So, let us say the input was, let us say 000111, let us see what happens. So, the first we need to like, all these rules, all these four rules you are replacing something from the stack. So, we can, right now the top of the stack is A, so you have to use the first two rules.

So, if you use, so in order to read, in order to process this you need to use the first rule. So, let us say we replace A with 0A1. Then what you can do is see the top of the stack is 0, so now you can read 0 from the input and then move to the next position, so you can read 0, and then you move to the next position. And 0 is removed from the top of the stack, but again the top of the stack is A, so now you can replace A with 0A1 again.

Now again the top is 0, so now you replace 0, sorry, you remove 0 from the input and you remove zero from the top of the stack. So, basically you used the first rule here once, then you use this rule, then you use this rule again and then you use this rule again. Now again there is one more 0, so to get that 0 we need to replace A with 0A1 once again, and then again use the third rule, that is 0 or you read from the input, and then 0 you pop from the stack.

So, now you have again A, but now you are not interested in 0s, because the input contains three 1s. So, now you can use the second rule, that A goes to empty string. So, which means you just pop A from the stack by this rule. Now we have three 1s and you can successively read them 1 by 1. And, as you successively read them you pop them off, so then you have read the entire input. But that is not enough.

So, now you have read the entire input but that is not enough, you are still in the middle state, you are not moved to the accepting state. but then the only way to go to the accepting state is to reach the dollar symbol, so now you have cleared whatever was there in the stack and you reached the dollar symbol, and now you can pop the dollar symbol and move to the accept state. So, now this means that the PDA accepts 000111.

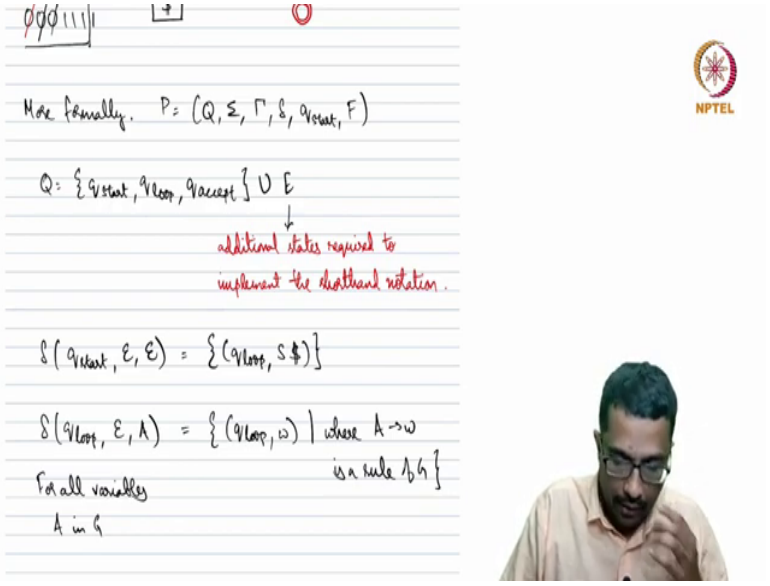
Now suppose the string was 0001111, this would not be accepted because you will be able to go to the accept state after reading the first six symbols, but the last one will remain unprocessed and if you need to obtain this one, then it is not possible to obtain this one. So, what I am saying is that you will go to the accept state before reading the last one; that means the entire string has not been read by the input, entire string has not been read by the PDA.

And if you have some other thing like 0 1 0 or some other thing, which is not in the 0^n1^n and or 0^*1^* also, you cannot, you can try to work out this, you cannot process that string using this PDA. So, because only you have a bunch of 0s and then a bunch of 1s and they have to be equal, only then it will be accepted.

So, now this is simple enough grammar, which is just, simple enough grammar but this tells us how and this can be converted into a simple enough PDA. So, there are three states, first you push a dollar, then you push a variable A, then you have four rules here corresponding to the two variables, sorry, two rules in the grammar and two terminals. And then you have understate and there is you just move to the state by popping the dollar.

So, that is how you construct this, so basically just more formally I will just say the same thing again. So, I have written the entire thing in details here.

(Refer Slide Time: 20:39)

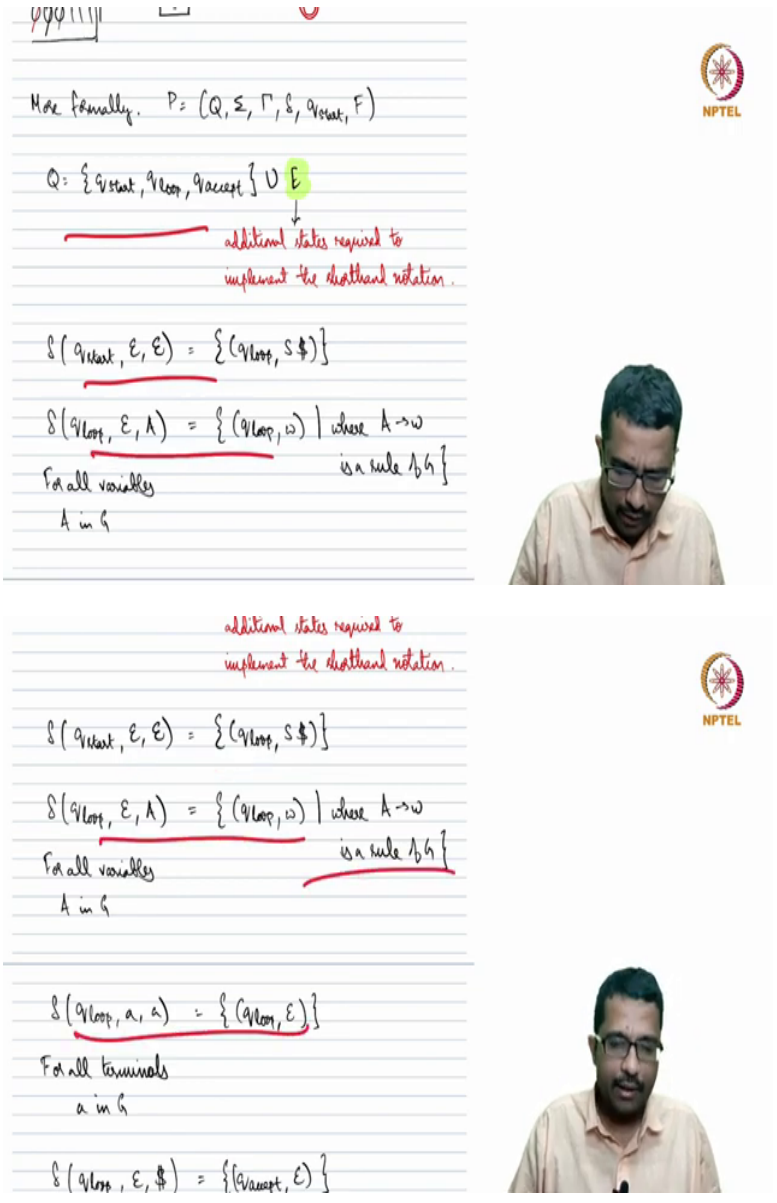


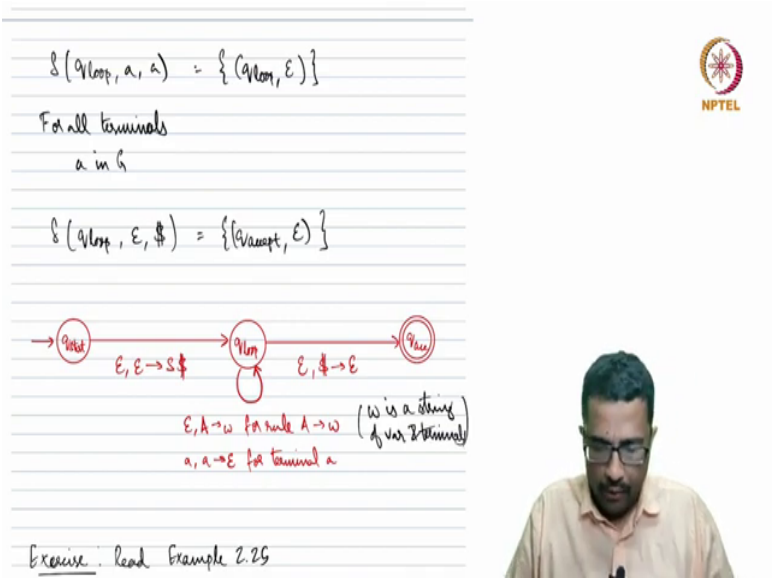
So, you have three states (q start, q loop and q accept) and in fact, you may have more states, why because we are using this shorthand notation, so the 0A1 is not a single string, A dollar

is not a single string, so we are using this notation that we just explained at the beginning which is this, so s goes to x y z. So, we need to add intermediate states to realize this.

So, for this A dollar we may need to add one intermediate state, for 0A1 we may need to add one or two intermediate states. So, the total number of states may be more than three depending on how many you add, but it does not matter, we can add some states.

(Refer Slide Time: 21:26)





And here I have written all the transition rules, so there is the starting to the next one and this is corresponding to all the rules, this is corresponding to all the terminals and finally, the transition from the middle state to the accept state. So, what are the rules again, you have three states, start, loop and accept, Q start, Q Loop and Q accept. Initially you push a dollar and the start variable and ending is you just pop a dollar to go from Q loop to Q accept.

The main set of rules come in the middle, this one. So, in Q loop that you do one of two things, one is if the top is a variable you replace it with a substitution rule of the grammar, so you may, the w here is a string of variables and terminals. W is a string of variable and terminals. And so that how many other rules you have that many ever, that many rules are there, that many such transition rules are here.

And for each terminal in the grammar, you have a rule where you read the terminal A from the input and you pop that terminal A from the top of the stack. So, these are the set of rules here. So, that the transition rule, so you have, if you have 10 rules in the grammar, there will be 10 rules of the first type; and if you have three terminals you have three rules of the second type, so 10 plus 3, 13 rules will be there for this loop.

And that is pretty much it; that is the construction for the PDA. So, that accept the same language as the context free grammar. So, you had the context free grammar G and whatever the rules are, now by the construction it should be clear that whatever string it accepts must be generated by the CFG and whatever is generated by the CFG must be accepted. So, nothing more nothing less.

So, in the book there is this example 2.25 that you can read, just this is another example on how you construct this. But the basic premise is simple, it is just these three rules, and then the Q loop has a bunch of rules depending on the grammar and the terminals. So, that is it. The goal was to show that context free grammars and PDAs are equivalent. And that is a theorem 2.20 here.

So, the first part is to show that anything that is context free there is an equivalent PDA, which is what we showed today, the meaning showed in this lecture. And for given a grammar we constructed an equivalent PDA. What is left which will be covered in the remaining, in the upcoming lectures is the other directions of the proof.

So, we will show that if a language is recognized by a PDA, it is necessarily context free, meaning we can build a grammar for it. So, that will be, that we will see in the next lectures. And that is it for lecture number 22. So, see you in lecture number 23. Thank you.