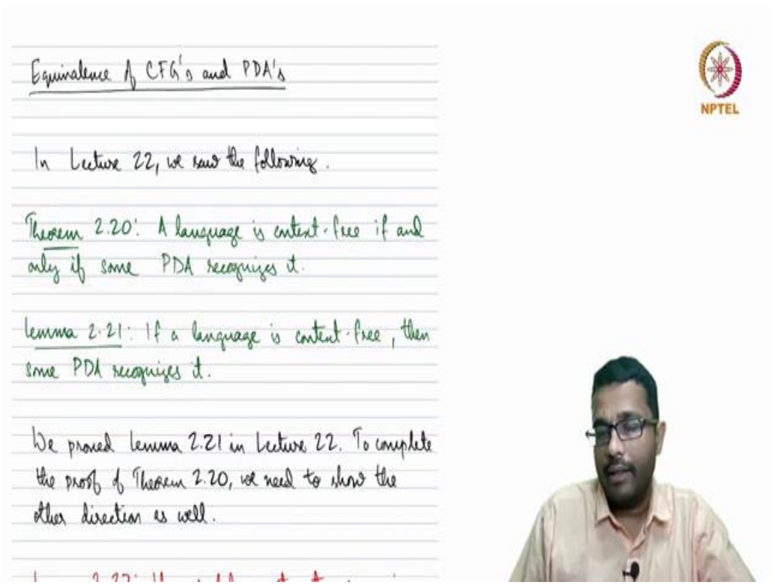
# Theory of Computation
**Professor. Subrahmanyam Kalyanasundaram**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Hyderabad**
**Equivalence of Context Free Grammars and Pushdown Automata - Part 02**

(Refer Slide Time: 0:17)



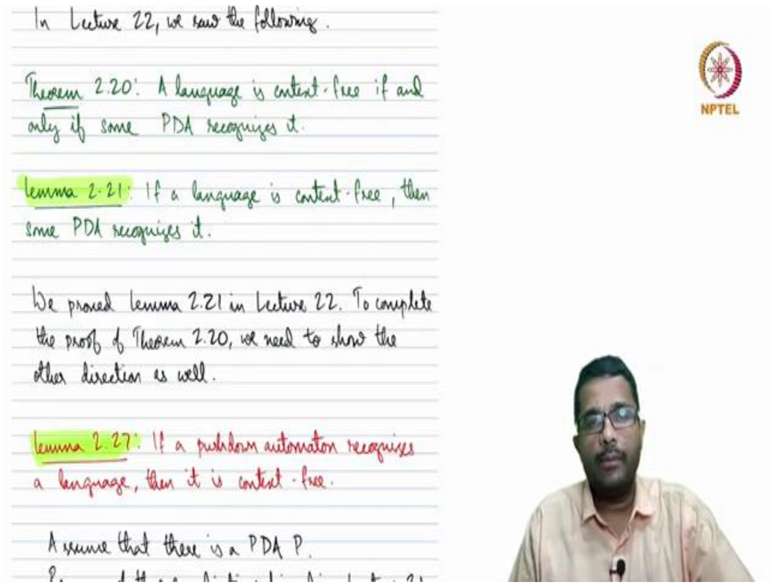
Equivalence of CFG's and PDA's

In Lecture 22, we saw the following.

Theorem 2.20: A language is context-free if and only if some PDA recognizes it.

Lemma 2.21: If a language is context-free, then some PDA recognizes it.

We proved lemma 2.21 in Lecture 22. To complete the proof of Theorem 2.20, we need to show the other direction as well.



In Lecture 22, we saw the following.

Theorem 2.20: A language is context-free if and only if some PDA recognizes it.

Lemma 2.21: If a language is context-free, then some PDA recognizes it.

We proved lemma 2.21 in Lecture 22. To complete the proof of Theorem 2.20, we need to show the other direction as well.

Lemma 2.27: If a pushdown automaton recognizes a language, then it is context-free.

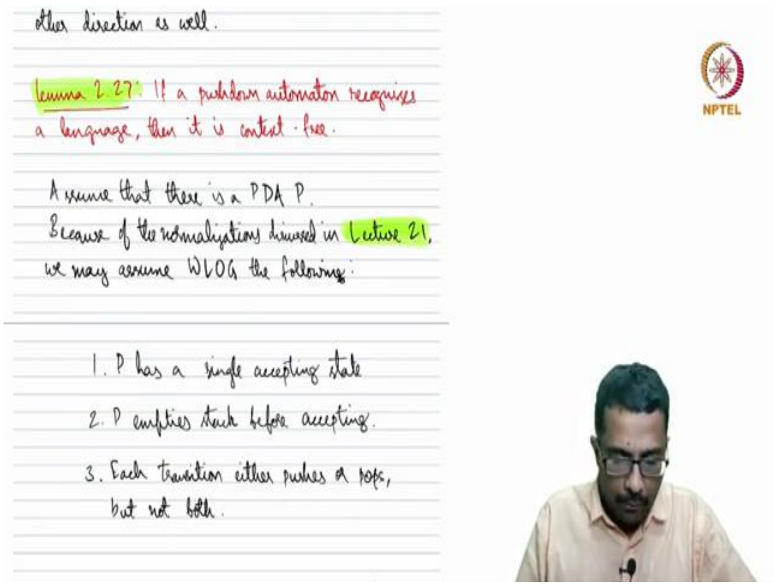
Assume that there is a PDA P.

Hello and welcome to lecture 23 of the course Theory of Computation. In lecture 22, we started to discuss the equivalence of context free grammars and pushdown automata. In this lecture we continue the proof of the same fact. So, we stated the following in lecture 22. One is the theorem. The main theorem is that a language is context free if and only if some PDA recognizes it.

And then we proved Lemma 2.21, which is one direction of the above theorem, which is that if a language is context free, there is some PDA that recognizes it. We completed this proof. Given a context free grammar we constructed, we explained how to construct a PDA that is equivalent. So, that is how we proved Lemma 2.21. So, that constitutes one half of the proof of theorem 2.20. Theorem 2.20 is an 'if and only if statement' so it needs both directions of the proof.

So, what is the next direction? The next direction says that if there is a pushdown automata that recognizes a certain language, then that language is context free. So, that completes, so this is Lemma 2.27; and Lemma 2.27 and Lemma 2.21 together constitute the proof of theorem 2.20. So, these two together constitute the proof of theorem 2.20. So, what we now have to show is that if a PDA recognizes a language, then that language is context free.

(Refer Slide Time: 2:01)

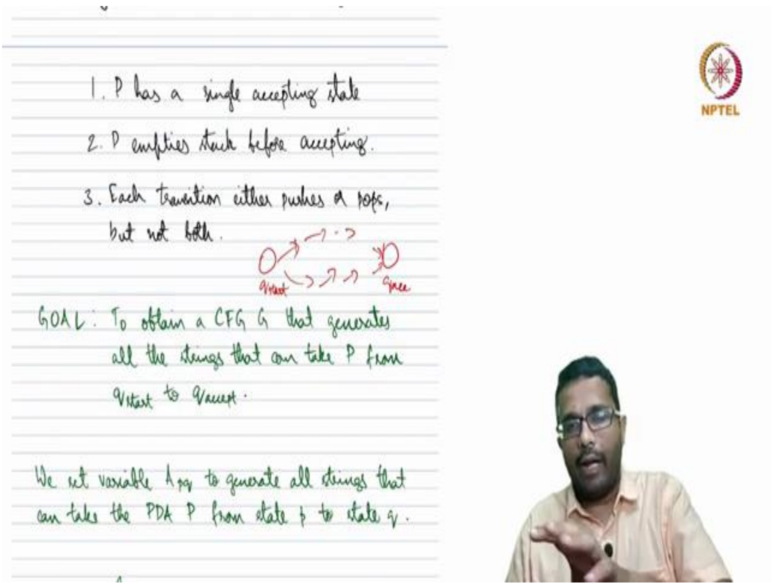


other direction as well.

Lemma 2.27: If a pushdown automaton recognizes a language, then it is context-free.

Assume that there is a PDA P.
Because of the normalizations discussed in Lecture 21, we may assume WLOG the following:

1. P has a single accepting state
2. P empties stack before accepting.
3. Each transition either pushes or pops, but not both.



1. P has a single accepting state
2. P empties stack before accepting.
3. Each transition either pushes or pops, but not both.

GOAL: To obtain a CFG G that generates all the strings that can take P from $q_{start}$ to $q_{accept}$.

We set variable $A_{pq}$ to generate all strings that can take the PDA P from state p to state q.

So, the way we do this is a kind of natural thing. If there is a PDA that recognizes a certain language, so there is a PDA, so given the PDA we will explain how to come up with a grammar such that this grammar will be equivalent to the PDA, meaning any string that is accepted by the grammar will be accepted by the PDA and anything that is not generated by the grammar will not be accepted by the PDA.

So, this will be equivalent, it will be exactly the same set of strings that is accepted by the PDA, which will be generated by the grammar. So, that is the kind of high level picture of what we are going to do. So, now let us get into the details of the same. So, the first point is that, if you recall in the last lecture of week 4 and lecture 21, immediately after seeing PDA we had explained a couple of things.

One is that given any PDA if it has multiple accepting states you can construct an equivalent PDA, which has a single accepting state. So, you may recall that this was merely by: if it has multiple accepting states you add a new accepting state and you make $\epsilon$ transitions from all the old accepting states to the new accepting state. So, that is how we did the first one.

The other two facts that we said is that given a PDA we can construct an equivalent PDA that always empties the stack before accepting. So, if the original PDA did not empty the stack before accepting, we can construct an equivalent PDA, where we can force it to empty the stack before accepting. So, this involved putting some symbol at the beginning of the stack and then ensuring that symbol is removed before it is accepted.

And the third thing is that we can ensure that every transition involves a push or a pop but not both. So we are saying that every transition either pushes into the stack or pops out of the stack, but does not do both. So, there will not be a transition of the type which does not touch the stack at all. There will also not be a transition which does both push and pop. So, every transition that you take will have a push or a pop, but will not have both.

So, now because we saw these things in lecture 21 and we explained how we can accomplish this, we will be using these things. So, the task is to construct a grammar that is equivalent to a given PDA. Now that we have told or we have seen how any PDA can be converted into a PDA with these properties, we may as well assume that the PDA that accepts the language also satisfies these properties.

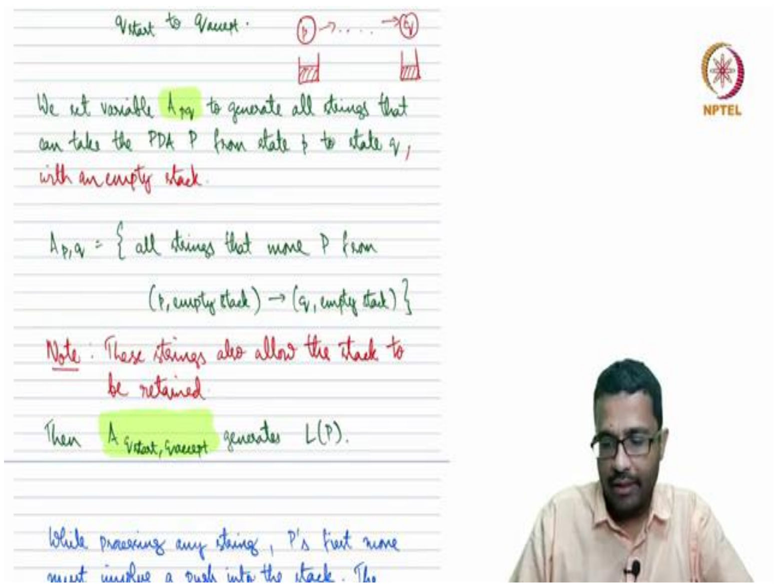
So, we may assume that the given PDA satisfies these properties, meaning it has a single accepting state, it empties a stack before accepting and every transition has a push or a pop. Now from such a PDA we will construct an equivalent grammar. So, what are the strings that are accepted by this PDA? These are the strings. So, there is a starting state to the PDA. Let us say there is a starting state $q_{start}$ and there is an accepting state $q_{acc}$. And the strings may travel somehow and finally reach $q_{acc}$.

So, what are the strings that the PDA can read from the starting state or that the PDA can read, so that it can move from the starting state to the accepting state, so this is what we want to know. What are the strings that can take the PDA from the starting state to the accepting state? So, you noticed that I said the accepting state. So, we are already making use of the fact that the PDA has a single accepting state.

If it has multiple accepting states, then I cannot say: the accepting state. It may not be a unique one. And because of property 2, we can assume that the stack will be empty when we reach the accepting state. Because we assume property 2, we can say that the stack will be empty. So, our goal will be to identify the strings that can take the PDA from the starting state to the accepting state.

And when it takes it to the accepting state the stack will be empty. Of course, when you start processing also the stack is empty, when you, the PDA does not come or does not begin operations with a filled stack. So, this is the key thing here, we want to know which are the strings that the PDA can read, which will allow it to move from the start state to the accepting state, and when it is reaching the accepting state the stack will be empty.

(Refer Slide Time: 7:51)


$q_{start}$ to $q_{accept}$.

$$\begin{tikzcd} \textcircled{p} \arrow[r] & \cdots \arrow[r] & \textcircled{q} \end{tikzcd}$$

We set variable $A_{pq}$ to generate all strings that can take the PDA P from state $p$ to state $q$, with an empty stack.

$A_{p,q}$ = { all strings that move P from (p, empty stack) → (q, empty stack) }

Note: These strings also allow the stack to be retained.

Then $A_{q_{start}, q_{accept}}$ generates L(P).

While processing any string, P's first move must involve a push into the stack. The

So, this turns out to be quite convenient in the construction that follows. So, what our main thing is going to be. So, now the goal is to construct a grammar such that the grammar should generate all these things that take the PDA from the start state to the accept state. So, what we will do is in general we will have variables of this type $A_{pq}$ which are the variables of the grammar. We will construct the grammar in such a way that this variable will generate all the strings that take the PDA from the state $p$ to the state $q$.

We will construct the rules of the grammar in such a way that $A_{pq}$ will generate all the strings that can take the PDA from the state $p$ to the state $q$ keeping the stack empty or with an empty stack. So, when it takes it to the state $q$, the stack will be empty. So, I am writing it here again, more formally. This is the set of all strings.

$$A_{pq} = \{ \text{all strings that move P from (p, empty stack)} \rightarrow \text{(q, empty stack)} \}$$

All strings that take the PDA, capital P from, the state $p$ and an empty stack to the state $q$ and an empty stack. So, this will be the set of all strings. Maybe I will just draw a small picture. Suppose this is state $p$, let us say the stack is empty here and you want these strings that can move from the state $p$ to state $q$. But notice this, from empty stack to empty stack, which means maybe something gets added into a stack and finally those gets removed.

Which means if the stack had some content here, let us say I am just denoting by the shaded area, if a string can take the PDA from $p$ to $q$ on empty stack, it can also take from $p$ to $q$ while retaining the contents of the stack. But how does empty stack to empty stack happen? Because it cannot pull out anything from the stack. Instead, it will push some things and by the time it reaches $q$ it retrieves those things.

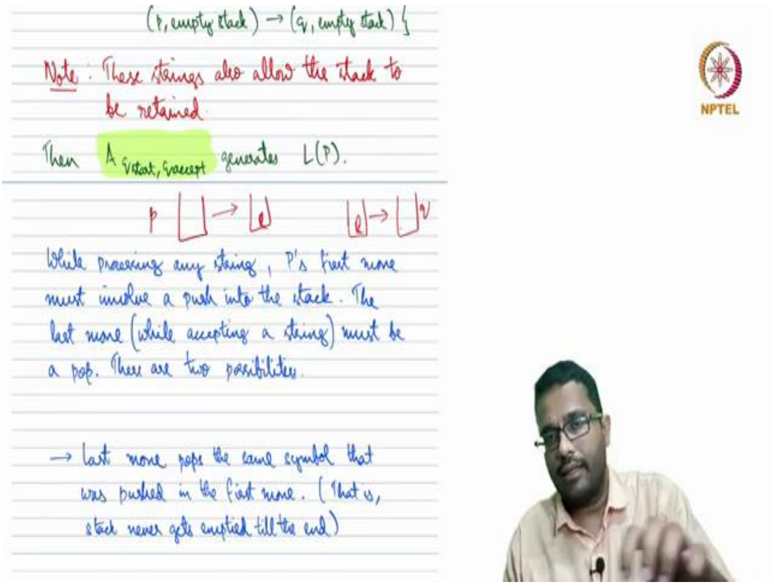
So, suppose the stack was not empty. It had some content. Let us say some string called k or something. So now, keeping the string k there, if you take the same set of transitions, it will put some things into the stack. So, originally it could take from empty to empty, so the k will be there and it will put some things in and then it will remove those things. When it reaches q, whatever was put would have been removed.

So, empty stack to empty stack also means that it can retain whatever the stack contents were, so this is something that will become relevant later. So, right now I am just defining it as those strings that can take the PDA from the state p with the empty stack to the state $q$ with the empty stack. But what I am remarking is that these strings also take from state $p$ to state $q$ while retaining the stack. So, maybe I will just remark here, these strings also allow the stack to be retained, so $A_{pq}$ also can be interpreted in this manner.

So, this is going to be for all any state $p$ and any state $q$ including where $p$ and $q$ are the same. So, now consider the variable $A_{q_{start}, q_{accept}}$. What are the strings that take the PDA from $q_{start}$, the starting state to the accepting state, while retaining the stack or while keeping the stack empty. This is exactly the set of strings that are accepted by the PDA, so that is what I have written here. $A_{q_{start}, q_{accept}}$, should basically generate those strings that are accepted by the PDA.

So, what we will do is that we will set this variable $A_{q_{start}, q_{accept}}$, as the starting variable. because what is the language generated by the grammar? It is the language set of strings generated by the starting variable. So, this will be our starting variable and this is the key thing. The variables are all of the type $A_{pq}$ and $A_{q_{start}, q_{accept}}$ will be, will generate those strings which are accepted by the PDA and that is what we want.

(Refer Slide Time: 13:53)



(p, empty stack) → (q, empty stack) {

Note: These strings also allow the stack to be retained.

Then $A_{q_{start}, q_{accept}}$ generates L(P).

While processing any string, P's first move must involve a push into the stack. The last move (while accepting a string) must be a pop. There are two possibilities.

→ last move pops the same symbol that was pushed in the first move. (That is, stack never gets emptied till the end)

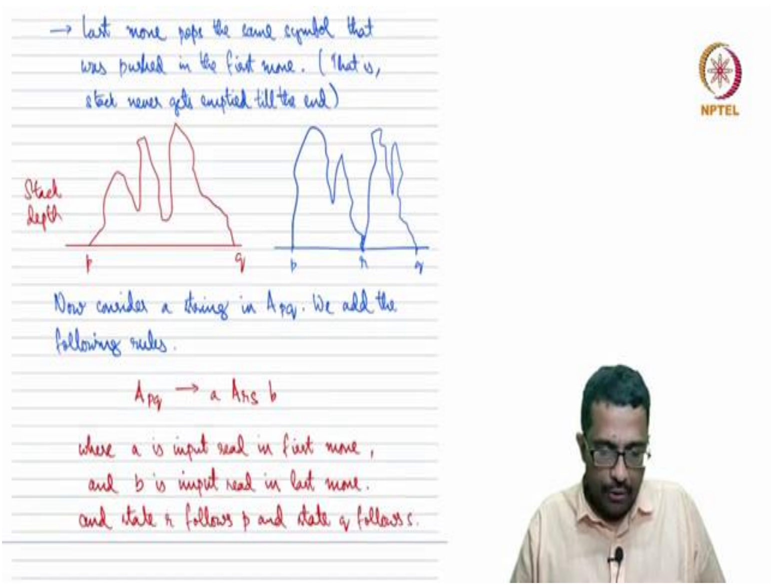
So, now let us try to understand the rules. So, we have just told what the variables are, so now we need to define the, we need to set up the rules of the PDA. So, let us consider what happens when we move from a state $p$ to a state $q$. So, when you are moving from a state $p$ to the state $q$, let us say we start with the empty stack. So, that is what the definition is, $(\text{p}, \text{empty stack}) \rightarrow (\text{q}, \text{empty stack})$. So, there are two possibilities.

So, let us see what happens. So, initially you are in state $p$ with an empty stack. We know that every move has a push or a pop. So, the next move from here must involve a push, because there is nothing to pop. So, the next move involves a push, let us say it puts some symbol, let us say r or something, let us say l or something into the stack.

And let us consider what happens, and at the need to go to state $q$ with the empty stack. Now what would have been the last move that took us to $q$. We know that every move has a push or a pop. The last move would not have been a push because if it was a push this stack will not be empty. So, the last move has to be a pop. So, some symbol, let us say k or something was there that was removed.

So, there are two possibilities; one is that the symbol that was pushed in the first move l is exactly the symbol that is popped at the end. Which means whatever we put on the stack at the beginning, now you kept putting things on top of it, maybe removing some things, etcetera, but we never touched the first symbol that was put. So, maybe I will just create some space here.

(Refer Slide Time: 16:27)



→ last move pops the same symbol that was pushed in the first move. (That is, stack never gets emptied till the end)

Now consider a string in $A_{pq}$. We add the following rules.

$$A_{pq} \rightarrow a\ A_{rs}\ b$$

where a is input read in first move, and b is input read in last move. and state r follows p and state q follows s.

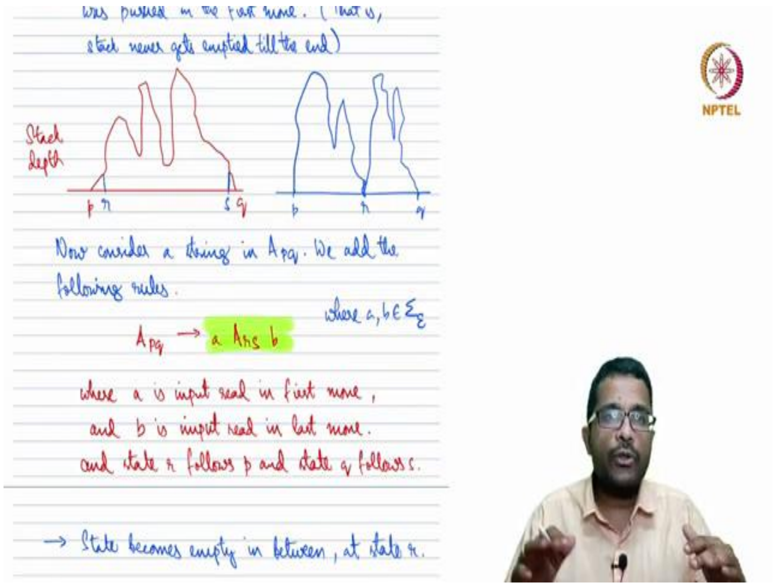
So, what I am saying is there are two possibilities, one possibility is that the first move you put something in and then on the stack put some things. So, here you are at state $p$ and at the end your state $q$. This is the stack content, stack depth or stack height or something you can call it. It does not have to be like this, it could go up, but it never empties in the middle, meaning the first symbol that you put, let us say k or something that or l or something that is never getting touched. So, this is one possibility.

Another possibility is, this is possibility two, the first symbol that we put, let us say k or l or something, something happened at some point you pop it. So, if you pop the first symbol that you put, then basically at some point the stack becomes empty and, so maybe let us say at some state, at that time we were in some state $r$ and then again you do something and finally the stack again becomes empty.

Which means some other symbol, the first symbol that we put was removed in the middle and then again, some things got put and even those things got removed. That is why you moved from empty to empty and again to empty, so these are the two possibilities. So, the red one and the blue one. So, first we will consider the red one and we will try to create a rule that sort of corresponds to the situation.

So, that is what I have written here. The last move pops the same symbol that was pushed in the first move, that is what I have written here, which means the stack never got empty in the meantime. It was always non-empty. So, now consider a string, so basically this rule, so it may not be immediately clear, but this rule is kind of corresponding to that.

(Refer Slide Time: 18:41)



was pushed in the first move. (That is, stack never gets emptied till the end)

Now consider a string in $A_{pq}$. We add the following rules.

where $a, b \in \Sigma_\varepsilon$

$A_{pq} \rightarrow a\, A_{rs}\, b$

where a is input read in first move, and b is input read in last move. and state r follows p and state q follows s.

→ State becomes empty in between, at state r.

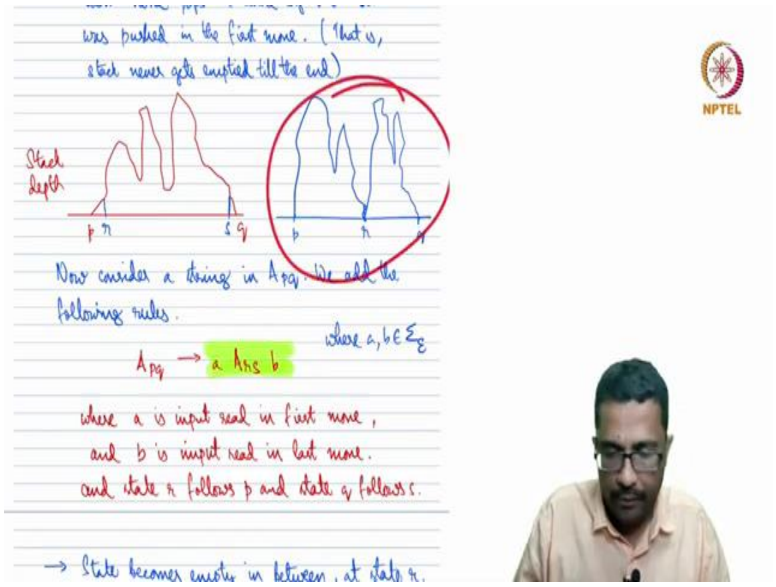
So, consider a string in $A_{pq}$, so remember a string generated by $A_{pq}$. Here I am writing it as a set, but it is a variable. So, now for this particular situation we add a rule like this, so where $A_{pq}$ derives or $A_{pq}$ yields this following thing. The rule is $A_{pq} \rightarrow a\, A_{rs} b$ where, $a, b \in \Sigma_\epsilon$ , which could be terminals or could be empty also.

And if you, basically you are able to read the symbol a and go to the state $r$ and then go from, then you have the variable $A_{rs}$ and then from $s$ you are able to read the symbol $b$ and then go to the state $q$. So, we are saying what is the state in the next step, let us say it is $r$ and what is the state in the last step before $q$, let us say it is $s$. Basically, what you are saying is the first symbol that was pushed here from $p$ to $r$ was something.

While we push that push that symbol. Let us say the symbol that was pushed was $t$. While we push that symbol, we read the input symbol $a$, and now at the end we again, we are again popping the symbol from when we move from $s$ to $q$, this the first symbol that I have pushed and that happens while reading the input $b$. Which means from $r$ to $s$ whatever was the stack that that single symbol, we never touched that.

So, the remaining part of the middle of the string took the took the PDA from $r$ to $s$ while keeping the stack intact, meaning whatever was already there in the stack it retained it. Which means that it can also take the PDA from $r$ to $s$ with an empty stack.

(Refer Slide Time: 21:19)



was pushed in the first move. (That is, stack never gets emptied till the end)

Stack depth

p r s q

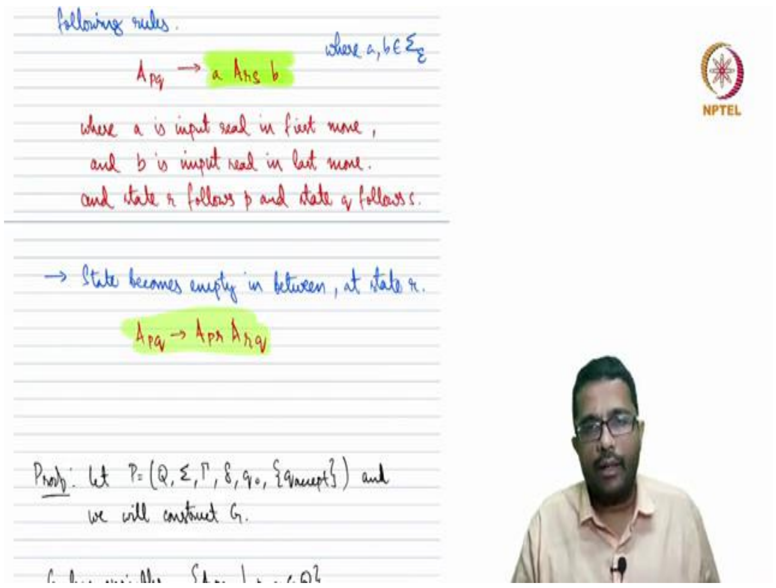
p r q

Now consider a string in $A_{pq}$. We add the following rules.

$A_{pq} \to a A_{rs} b$ where $a, b \in \Sigma_\varepsilon$

where a is input read in first move,
and b is input read in last move.
and state r follows p and state q follows s.

→ State becomes empty in between, at state r.



following rules.

$A_{pq} \to a A_{rs} b$ where $a, b \in \Sigma_\varepsilon$

where a is input read in first move,
and b is input read in last move.
and state r follows p and state q follows s.

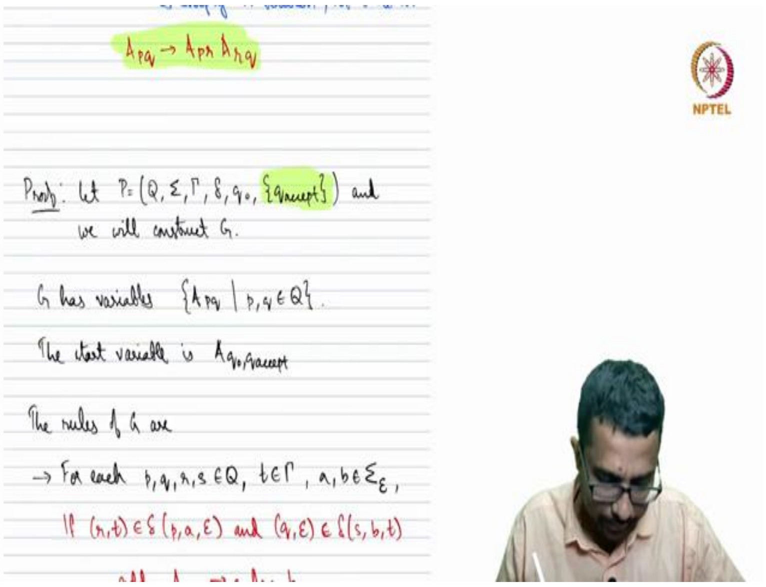
→ State becomes empty in between, at state r.

$A_{pq} \to A_{pr} A_{rq}$

Proof: Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and we will construct G.

And this is the first type of rule, and the second type of rule is corresponding to the blue part over here. This one where the stack gets empty. So, for that we add the following type of rules, where for every three variables $p$, $q$ and $r$, we add the rule $A_{pq} \to A_{pr}A_{rq}$. And for all triplets of rules, states p, q and r. I am sorry, there is an error. It is $A_{rq}$, so not $A_{rs}$. So, for all triplets of states $p$, $q$ and $r$ we define this.

(Refer Slide Time: 22:18)



$A_{pq} \to A_{pr} A_{rq}$

Proof: Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and we will construct G.

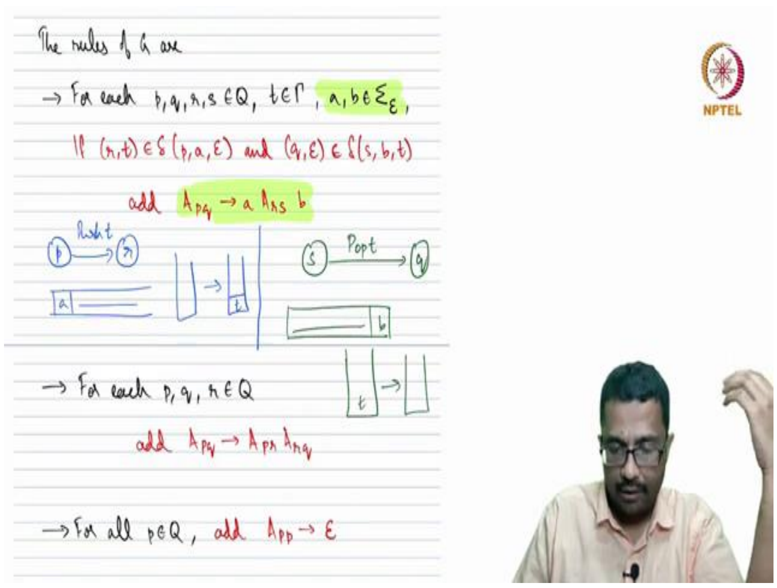
G has variables $\{A_{pq} \mid p, q \in Q\}$.

The start variable is $A_{q_0, q_{accept}}$

The rules of G are

→ For each $p, q, r, s \in Q$, $t \in \Gamma$, $a, b \in \Sigma_\varepsilon$,

If $(r,t) \in \delta(p,a,\varepsilon)$ and $(q,\varepsilon) \in \delta(s,b,t)$



The rules of G are

→ For each $p, q, r, s \in Q$, $t \in \Gamma$, $a, b \in \Sigma_\varepsilon$,

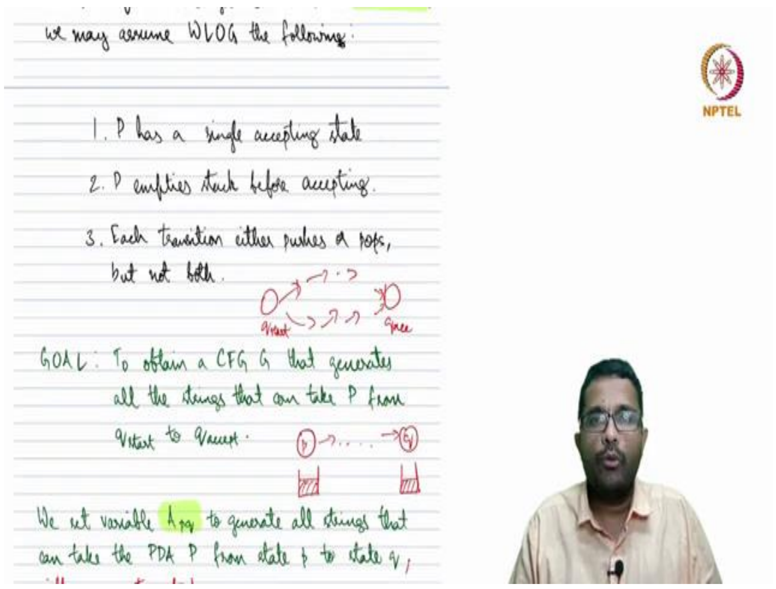
If $(r,t) \in \delta(p,a,\varepsilon)$ and $(q,\varepsilon) \in \delta(s,b,t)$

add $A_{pq} \to a A_{rs} b$

→ For each $p, q, r \in Q$

add $A_{pq} \to A_{pr} A_{rq}$

→ For all $p \in Q$, add $A_{pp} \to \varepsilon$



we may assume WLOG the following:

1. P has a single accepting state
2. P empties stack before accepting.
3. Each transition either pushes or pops, but not both.

GOAL: To obtain a CFG G that generates all the strings that can take P from $q_{start}$ to $q_{accept}$.

We set variable $A_{pq}$ to generate all strings that can take the PDA P from state p to state q, So, maybe I will just say it a bit more formally here. And this is the intuition, so the intuition I wanted to convey before that. So, more formally, suppose there is a PDA with a single accepting state. So, notice that we are able to write a single accepting state because of the normalization that we refer to here. P has a single accepting state. And we are able to say that these things, like the first symbol, the first move should be a push because of the second condition.

Because we are insisting that every move has to be a push or a pop otherwise the first move could have been nothing. So, the variables are all of the type $A_{pq}$ and the start variable is $A_{q_0, q_{accept}}$. So, instead of $q_{start}$ I am just using $q_0$. And the rules are this. So, once again let me explain, for any four states, $p, q, r, s \in Q$ and $t \in \Gamma$. So, $t$ is a symbol in the stack alphabet and $a, b \in \Sigma_\epsilon$.

$$\text{if } (r, t) \in \delta(p, a, \epsilon)\ and\ (q, \epsilon) \in \delta(s, b, t)$$
$$\text{add } A_{pq} \to a\ A_{rs} b$$

Which means a, b could be symbols of the input alphabet. Input alphabet not stack alphabet. They could also be empty strings. So, what are we saying here? We are saying that if $(r, t)$ is in $\delta(p, a, \epsilon)$ meaning you have the input, so the next symbol from the input is $a$ and something else and you are in state $p$, so you are able to read $a$ and move to state $r$ while pushing $t$ and you are popping nothing. So, $\epsilon$ is popped.

So, from empty stack, you will go to a stack with a symbol $t$. If this is a rule and the next part is, $(q, \epsilon) \in \delta(s, b, t)$, meaning from the state $s$ you are able to pop t and go to q and you have read some parts of the input and the symbol that you are now reading is symbol $b$. And you are able to move from, basically you have, let us say you have $t$ in the stack, you want to remove the $t$ in the stack.

Basically, you are popping $t$ from the stack and you are pushing nothing. Then both of these things are satisfied, if you can go from $p$ to $r$ by pushing $t$ and reading symbol $a$ and you can go from $s$ to $q$ by popping $t$ and reading symbol $b$; then you add this rule, $A_{pq} \to a\ A_{rs} b$. Notice that $p, q, r, s$ need not all be distinct, $a$ and $b$ could be empty strings. But $t$'s cannot be empty string because every move has to have a push or a pop. $t$ will not be empty.

So, this corresponds to the case where the symbol that we first pushed is popped at the end. Once again, if you are able to move from $p$ to $r$ by reading the symbol $a$ from the input and pushing $t$ into the stack and you are able to move from $s$ to $q$ by popping $t$ from the stack and reading $b$ from the input state, then you add these rules, add these rules of the type, $A_{pq} \rightarrow a\ A_{rs}b$. So, that is the first set of rules.

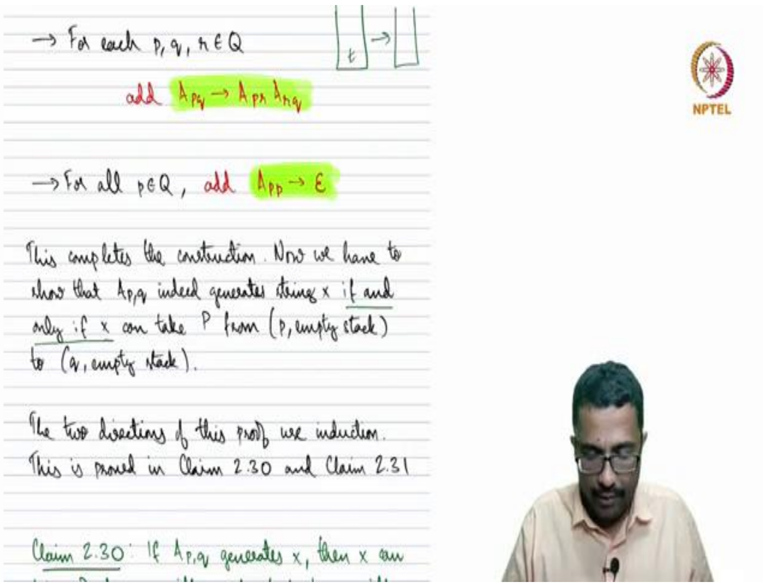
And that you have to do for all the possible quadruples $p, q, r, s,$ whenever we are able to identify such, we have to see whether these rules are there. If we can say that $(r, t) \in \delta(p, a, \epsilon)$ and so on, then we add this rule. The next set of rules are simpler, because there is no condition to be checked. For any triplet of states $p, q, r$, again including the case where they are not distinct, we add the rules $A_{pq} \rightarrow A_{pr}A_{rq}$.

So, it is straightforward $A_{pq} \rightarrow A_{pr}A_{rq}$. We add these rules for all the triplets $p, q\ and\ r$. And finally, if you notice if you just have these two types of rules, the type 1 and type 2 that I mentioned, you notice that every rule has a variable in the right hand side. The starting variable was $A_{q_0,\, q_{accept}}$ and all the rules are of this type. It has a variable in the hand side, so it is never going to generate a string that is entirely of terminals.

So, you need to have rules where somehow there is no terminal in the right hand side or somehow we have to eliminate the terminal, no variable in the hand side or we have to eliminate these variables. So, for that we add these rules, for all the states $p$ in the PDA we add a rule $A_{pp} \rightarrow \epsilon$. So, notice that the third set of rules is consistent with what we want. We wanted the rules $A_{pp}$ to generate all the strings that can take the PDA from a from state…

So, $A_{pq}$ we wanted it to be the set of all strings that can take the PDA from $p$ to $q$ on an empty stack. So, empty string takes the PDA from $p$ to $p$ on an empty stack, it trivially does it. Empty string mean it reads nothing and you do not make any move. If you are already at an empty stack you will continue to be at an empty stack. So, that way this is kind of a base case.

(Refer Slide Time: 29:15)



→ For each $p, q, r \in Q$

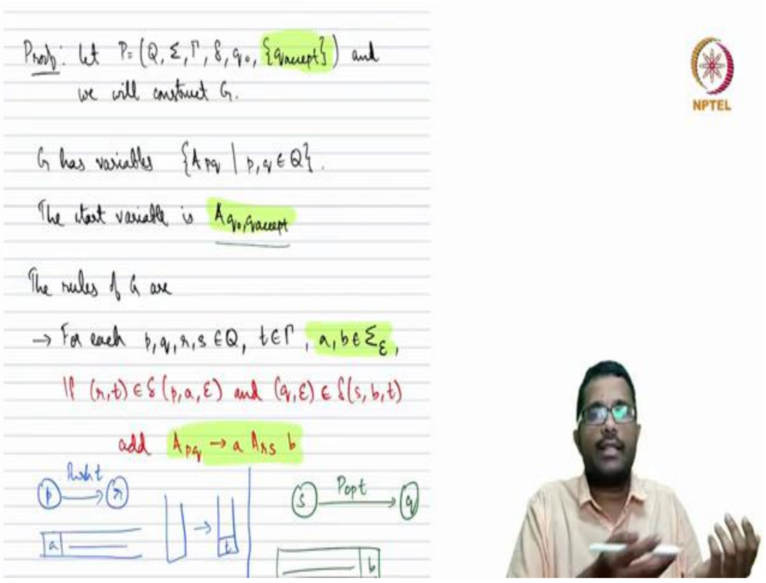
add $A_{pq} \to A_{pr} A_{rq}$

→ For all $p \in Q$, add $A_{pp} \to \varepsilon$

This completes the construction. Now we have to show that $A_{p,q}$ indeed generates string x if and only if x can take P from (p, empty stack) to (q, empty stack).

The two directions of this proof use induction. This is proved in Claim 2.30 and Claim 2.31

Claim 2.30: If $A_{p,q}$ generates x, then x can



Proof: Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and we will construct G.

G has variables $\{A_{pq} \mid p, q \in Q\}$.

The start variable is $A_{q_0, q_{accept}}$

The rules of G are

→ For each $p, q, r, s \in Q$, $t \in \Gamma$, $a, b \in \Sigma_\varepsilon$,

If $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$

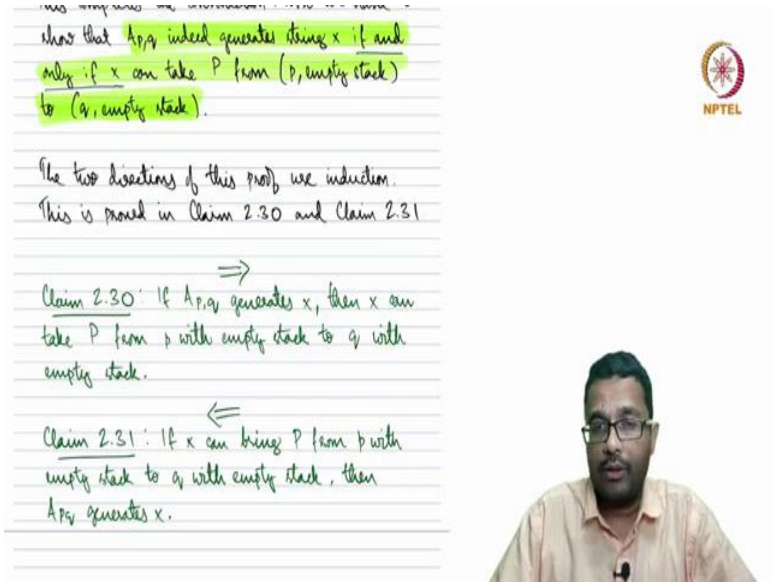
add $A_{pq} \to a A_{rs} b$

So, this completes the construction. And the way we have constructed it perhaps may not be immediately clear that this generates the grammar that is constructed in this manner is equivalent to the PDA. So, we want to show that for all $p$ and $q$, $A_{pq}$ generates a string $x$, if and only if $x$ can take the PDA P, from the state $p$ with an empty stack to the state $q$ with an empty stack.

So that is, once we show that the rest becomes clear. Because if you show that for all the states, all the variables $A_{pq}$ we want to show that $A_{pq}$ generates a string $x$ if and only if $x$ can take the PDA from p and empty stack to q and empty stack. If we show this then it follows that the starting variable $A_{q_0, q_{accept}}$. This generates all the strings that can take from the starting state to the accepting state on an empty stack. Which means that the set of strings accepted by the PDA is equal to the set of strings generated by the grammar.

So, this is the key thing that we have to show from now on in order to prove that this construction results in a grammar that is equivalent to the PDA. So, this again, in fact, we are in one direction of the proof of the theorem. Now we are again in a situation where we have to prove this equivalence that the constructed grammar is equivalent to the PDA or the constructed grammar, this variable generates exactly the string that we want.

(Refer Slide Time: 31:12)



show that $A_{p,q}$ indeed generates string x if and only if x can take P from (p, empty stack) to (q, empty stack).

The two directions of this proof use induction. This is proved in Claim 2.30 and Claim 2.31

$\Rightarrow$

Claim 2.30: If $A_{p,q}$ generates x, then x can take P from p with empty stack to q with empty stack.

$\Leftarrow$

Claim 2.31: If x can bring P from p with empty stack to q with empty stack, then $A_{pq}$ generates x.

So, what we will do is to show the following two claims, the first claim is that if $A_{pq}$ generates $x$, then $x$ can take the PDA from $p$ with an empty stack to $q$ with an empty stack. So, this is one direction and the other direction is that if string $x$ can take the PDA from $p$ with an empty stack to $q$ with an empty stack, then $A_{pq}$ generates $x$. So, together these two claims will prove this highlighted sentence.

The part that I am highlighting now $A_{pq}$ generates the set of all strings that can take the PDA from $(p, empty\ stack)$ to $(q, empty\ stack)$. So, claim 2.30 and 2.31 together will prove this. Since I think we are kind of over the usual, the prescribed time limit, we will show the proofs of 2.30 and 2.31 in the next lecture.

So, just to summarize what we saw - the goal was to prove the direction of the proof, which says that if PDA recognizes a language, then that is context free. The approach to proving it was to construct a grammar given a PDA. So, we assumed some normalizations, which we had shown that was possible for any PDA. Namely PDA has a single accepting state, a PDA empties stack before accepting and every transition involves a push or a pop but not both.

And the grammar involved variables $A_{pq}$ which were corresponding to those strings that can take the PDA from state $p$ with an empty stack to state $q$ with an empty stack. And we constructed the rules of the grammar. And now what remains is to show that the variables $A_{pq}$ indeed generate exactly those set of strings that are desired. And that will be proved by the following claims 2.30 and 2.31. And this we will see in our next lecture. So, see you in the next lecture.