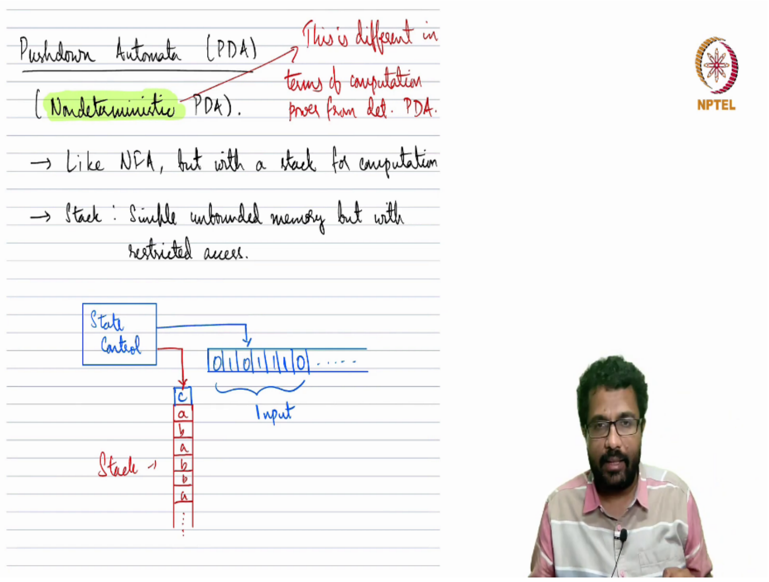
Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
An Introduction to Push Down Automata

(Refer Slide Time: 00:16)



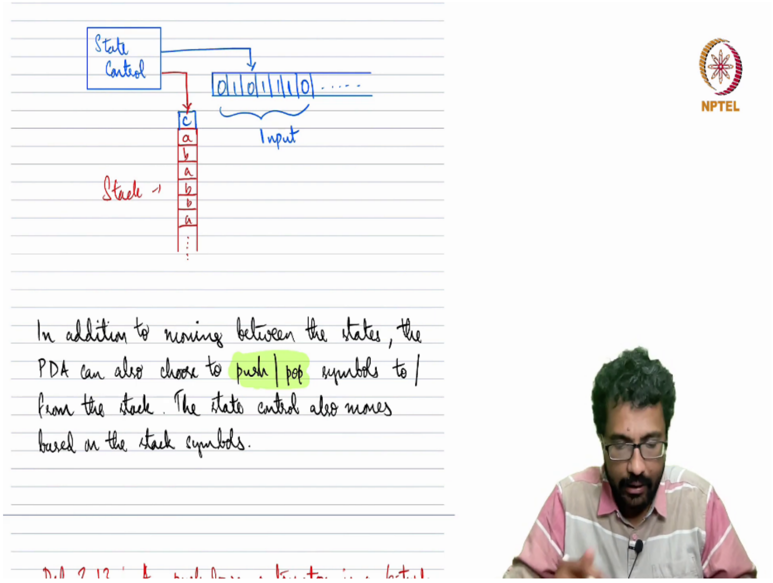
Pushdown Automata (PDA)

(Nondeterministic PDA).

This is different in terms of computation power from det. PDA.

→ Like NFA, but with a stack for computation

→ Stack : Simple unbounded memory but with restricted access.



In addition to moving between the states, the PDA can also choose to push | pop symbols to / from the stack. The state control also moves based on the stack symbols.

Hello and welcome to lecture 20 of the course Theory of Computation. In this lecture, we will see pushdown automata. In the previous lectures, we had seen context free grammars and the languages resulting from context free grammar, which is context free languages. So, in this lecture, we will see pushdown automata. In fact, what we are going to see is actually non-deterministic pushdown automata.

So, there is a the just like we had deterministic finite automata and non-deterministic finite automata, we have deterministic pushdown automata and non-deterministic pushdown automata but, unlike regular finite automata wherein find the case of finite automata both deterministic and non-deterministic finite automata have the same power. So, both recognize the class of languages which are regular languages over here that is not the case.

So, this is different in terms of computation power from deterministic pushdown automata. So, non-deterministic pushdown automata are different in terms of computation power meaning there are languages that we can recognize by using non-deterministic pushdown automata but we cannot recognize using deterministic pushdown automata. So, the non-determinism really adds some value, in other words in the case of finite automata you could convert a non-deterministic finite automaton to a deterministic finite automata here we cannot do that.

So, there are reasons however, in this course, we are not going to see the deterministic pushdown automata. So, whenever I say pushdown automata from now on it is going to be the non-deterministic one. So, we will not talk about deterministic pushdown automata at all and but then just for the completeness I would like you to know that there is something called deterministic pushdown automata and the deterministic pushdown automata has a slightly different computational power meaning there are languages that are recognized by non-deterministic pushdown automata which are not recognized by deterministic pushdown automata.

So, in terms of computation, this is very much like a NFA which is not a deterministic finite automaton. So, pushdown automata is also abbreviated by PDA pushdown automata. So, PDA's are like NFA's, but there is an additional stack for computation. So, the stack is like a data structure that we have learned, we may have learned in data structures. So, it is an basically it is an infinite, it is an ideal stack. So, it has infinite depth, it is an unbounded depth.

So, basically we can, there is no limit, it is not in a limited capacity limited memory, it is an unlimited memory, but it has there are restrictions on how we can access them. So, in terms in very simple terms, in the case of NFA, we had a. So, if you look at the figure, there is a tape that it contains some input 0101110. So, this is the input, this is the input and we just read the input and then make transitions as per the, the rules, that is what we did in the case of NFA.

In the case of DFA as well, but in the case of NFA there are more flexibilities, in the case of pushdown automata, or PDA's, we have this additional stack at our disposal. So, there is a stack which means you could let us say you could push or pop things from it. So, stack is like a you can think of it as a stack of plates or stack of books or something. So, you could keep adding things to the top of it.

But the restriction that you cannot remove things, you can remove the top book or the top plate, you cannot remove things from the bottom if you remove try to remove things from as like in a real life stack of books, then things get messy. But in the case of the data structure stack, you are not allowed to remove things from the bottom or the middle? Whatever is at the top, you can remove. And then once you remove that, you will see the next item, the one below the top, which is a new top. Now you can remove that and like that you can remove, right. So the stack also may have some symbols, a, b, a, b, b, something, so.

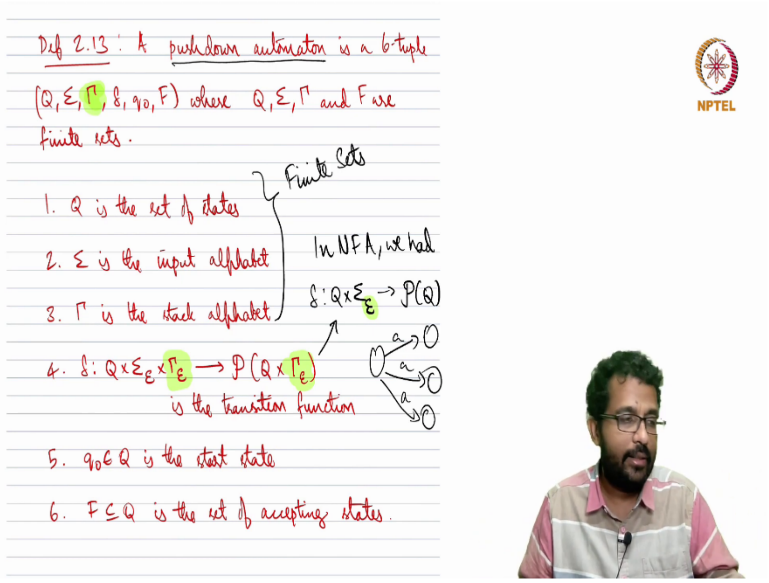
So now if you add, maybe just to give you an illustration, if you add a c on top of the stack, if you add a c on top of the stack, maybe I will use a different colour to denote that. So the c will go and sit on top of this? And if you want to remove the elements of the stack, you cannot remove from anywhere you will have to remove from the top. So this is the stack. And the key thing is that the stack is not a fixed memory.

So in theory, you could, there is no limit to how many items you could, how many symbols you can push down the stack? So that is the stack? So it is going to be an NFA with the stack. So that is what I have written here, in addition to moving between the states, so an NFA moves between the states. And in the case of pushdown automata, it can also push and pop symbols to the stack or from the stack. And in the case of NFA, we just saw the next symbol and then made the transitions, in the case of stack, sorry, in the case of PDA we may also look at the next symbol of the stack.

So in this in the figure that is about we say that the next symbol in the input is 0, and the top of the stack is c. So then we may say that now you push a new symbol into the stack, and then you make the transition q1 to q3 or something. So the transitions will depend on what you see at the stack, as well as what you see at the input. It can depend on what you see in the stack, it need not depend also.

So we could just make a transition based on the input, we could also make Epsilon transitions, we could also make a transition just based on the stack? So any combination is possible. So that is what I have written the second line the state control also moves based on the symbols seen from the stack.

(Refer Slide Time: 07:49)



Def 2.13: A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are finite sets.

1. Q is the set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet

Finite Sets

4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
5. $q_0 \in Q$ is the start state
6. $F \subseteq Q$ is the set of accepting states.

In NFA, we had

$\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$

So, I will just make the formal definition the formal definition is that a pushdown automaton, pushdown automaton that is singular automaton is a 6 tuple Q Sigma Gamma. So, this is new (Q ,Σ, Γ, δ, q_0, F). So, there Q ,Σ, Γ and F are finite sets. So, this is exactly the same as what we saw in the case of NFS non-deterministic finite automata, where you had Q sigma delta Q0 F also in the case of DFA but we did not have a gamma.

So, what is gamma is the alphabet for the stack. So, this need not be the same as the input alphabet you could use other symbols for the stack you could use a disjoint set of symbols if you need not use a disjoint set you could use the same set of symbols or a like a superset of subset or some combination is also. So, the Q is a set of states sigma is the input alphabet Γ is the stack alphabet. So, all of these are finite sets. Q0 is the start state just like in DFA or NFA and F is a subset of the set of states which is an accepting state.

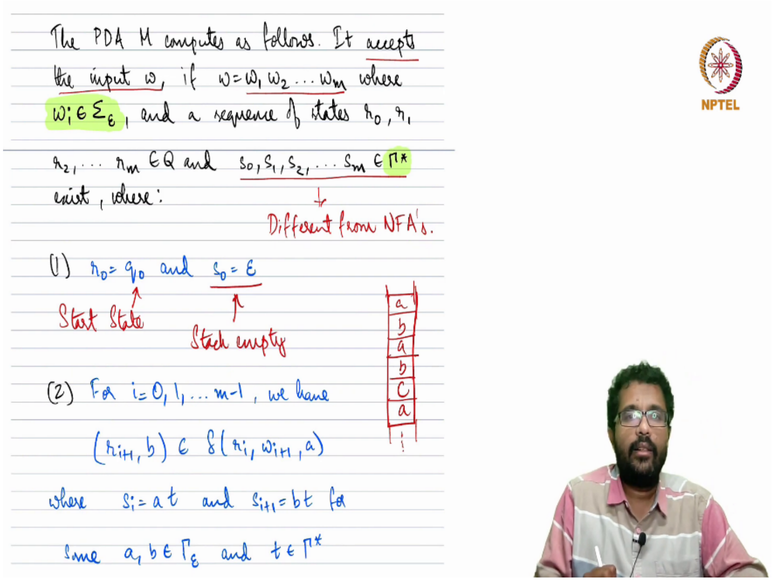
So, the main thing that is going to be different is the transition function. So, in the case of in the case of NFA the transition function was like this δ(Q x sigma) going to a set of states. So, in NFA. So, I can just make some space here, these are finite sets. So, in NFA we had this this transition function where, depending on what state you are in, and depending on what symbol you see as part of the input, and also epsilon, epsilon transitions are allowed. That is why we had this, we have the subscript epsilon, we can go to some subset of states?

So it could be like, so let us say the symbol a, you go to one state, or maybe two state, maybe three state, all of this is possible. In the case of, this is the case of NFA. In the case of PDA, the difference is this gamma here, gamma, subscript epsilon, and also this gamma here, this is where it differs, meaning the PDA could read the symbol on the top of the stack. So, this is the only symbol that it could access, it could read the symbol on the top of the stack. And then it could also write symbols on the top of the stack, or rather, it could push symbols from the stack. And it could sorry, it could push symbols onto the stack, it can pop symbols from the stack also.

So and the transition rules can be based on what symbol it sees, or what symbol it sees. So when it sees it pops a symbol, and then there are some transition rules. And the transition rules may result in a symbol being pushed onto the stack also. So it is like an NFA. But in addition to all the states and the symbols that you read, there is a new dimension, which is that what you read from the stack, again, we do not necessarily need to make reading from the stack at all transitions.

We do not necessarily need to write into the stack at all transmissions. But we can do both we can do, we can do neither, we can do one or four possibilities are there, we could just push without popping, we could just pop without pushing, we could both read and write or push and pop, we could do neither, we could just completely ignore the stack for a transition and just based on the input symbol.

(Refer Slide Time: 12:15)



The PDA M computes as follows. It accepts the input w, if $w = w_1 w_2 \dots w_m$ where $w_i \in \Sigma_\varepsilon$, and a sequence of states $r_0, r_1, r_2, \dots r_m \in Q$ and $s_0, s_1, s_2, \dots s_m \in \Gamma^*$ exist, where:

Different from NFA's.

(1) $r_0 = q_0$ and $s_0 = \varepsilon$

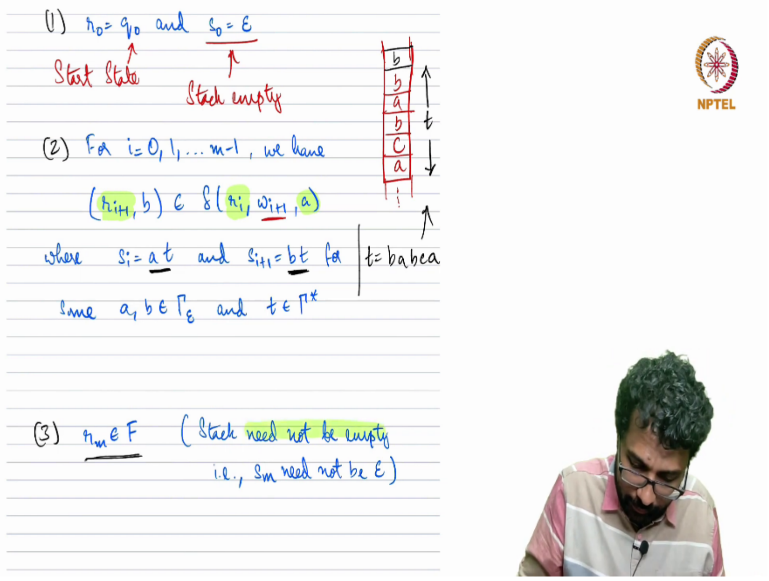
Start State

Stack empty

(2) For $i = 0, 1, \dots m-1$, we have

$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$

where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$



(1) $r_0 = q_0$ and $s_0 = \varepsilon$

Start State

Stack empty

(2) For $i = 0, 1, \dots m-1$, we have

$(r_{i+1}, b) \in \delta(r_i, \underline{w_{i+1}}, a)$

where $s_i = \underline{a}t$ and $s_{i+1} = \underline{b}t$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$

t= babca

(3) $\underline{r_m \in F}$ (Stack need not be empty i.e., s_m need not be ε)

So since we have seen NFA's, maybe it is not that difficult to see. So let me formally define what constitutes acceptance of a string in the PDA. So we say that the PDA M accepts the input w if we can write w as, like M symbols, w_1, w_2 w_m. So M symbols were, so I did not use. So usually we use N for the length of w. So the reason we are using M here is that each wi is in sigma subscript epsilon, meaning it could be empty string also.

So, like I could, for instance, I could write 1 0 1 0. Maybe I could write it as (1 0, ϵ) and 1 0 empty string. So because maybe I am using empty transitions in the middle. So this is also similar to what we had, we had exactly the same thing in NFA's, as well. And a sequence of states r0, r1 up to rm, this is also what we had NFA’s. The difference is this, this is the main difference, this is a new thing. This is where we knew sorry. We did not have this in NFA's.

So let us see what this means. So what is s_0, s_1 etcetera. And notice that these are these are part of gamma star, meaning these are strings that are formed using the stack alphabet. Which means actually, it is like this. So suppose the stack contains some, suppose at some point, the stack contains this string, a, b, a, b, c, a.? That is, it. And that is, that is the, that is the bottom. So now we will say at this stage, the stack, the string corresponding to the stack is a, b, a, b, c, a? And in the next step, I may add push a string, so let us say a push a.

So then the string in the stack becomes a, b, a, b, c, a. I could pop something from the stack. So, let us say pop to the top a, so then this thing in the stack becomes b, a, b, c, a? So, this is what I mean by string of the stack. So, and as usual the rules for acceptance are similar to before so, we should start correctly all the moves should be proper and we should end at an accepting state. So, this is the starting part, start correctly.

So, this means r_0 is a starting state. So, this is the start state and this means stack is empty to begin with these, start with the start state and then we say that the stack is empty and the next rule says that all the transitions should be proper meaning at the ith move or at the i+1th move. So, initially move one takes you from the 0 state which is the start state to the first state. So, we want so, the start the state before let us say the state before let us see the state before the rule the state before the ith step was r_i.

So, this is what I mean by ri and the state before is ri and suppose t is the string of the stack. So, when I say that what I mean is So, if you look at this figure that we have here I called if suppose this is what is there the stack, so, this is t so, in this figure t is a sorry, I will just write it separately t is a b a b c a because that is the string in the stack, suppose t is what is already there.

So, and ri is the state that we are in and the next symbol that we are reading is wi plus 1 so, is wi plus 1 and let us say the top part of the stack is a which is actually the case here the. So, maybe I should not have written t as this I should have written t as one level lower in everything, but the top symbols. So, t is sorry for the confusion t is b a b c a. So, the top symbol is not part of t.

So, this part basically the top it only the top part the changes. So, when you so, at this point you are at state ri, the next symbol of the input is wi plus 1 and the top of the stack is a right the top of the stack is a. So, that is how we have r_i w_i + 1 and a and then there could be multiple rules corresponding to this. So, notice that the rule is Q Sigma Gamma for a function of Q sigma gamma.

So, we have the status ri which is NQ sigma is w_ (i + 1) which is the next symbol, the stack is a, based on this we'll there are many possible things then, so, the for the next state and the next thing to be returning to the stack. So, suppose the, so what we want is. So, if the sequence of states are to be valid, then we should have r_i + 1 as a possible candidate here. So, ri plus 1 is the next state and b should be. So, suppose the symbol is b.

So, this is the symbol to be written into the stack because, if you remember in NFA there is nothing to be written because you only read the input, but here we have a stack onto which we can write things. So, where suppose this is the, this is the stack this is a rule. So now, which means before the transition the stack content was a followed by t because we saw a and the rest of the stack content, we are calling it t, we are calling it t and now a was read by this in this move and the and we wrote b.

So, now what happens is this a was erased and we wrote b. So, now, a t, a followed by t is now being changed to b followed by t. So, now, if at is replaced by bt this is what I have written here. So, before the ith transition si was at after the ith transition the stack content is bt. So, you are in state ri and you read the next symbol w_(i + 1) you read the top of the stack a, and one of the rules is that now you move to r_(i + 1) and you replace the top of the stack with b and therefore, you get that earlier it was a followed by the string t in the stack now, it is b followed by t.

So, this has to be so, this has to be met every move basically every time if we have ri wi plus 1 a which is the state next symbol of the next symbol of the input and the top symbol of the stack, then the successive state successor state ri plus 1 and b should be such that it should be part of the transition rule it should be it should be legal transition rule such that the sequence of the stacks, stack contents are also preserved or also maintained.

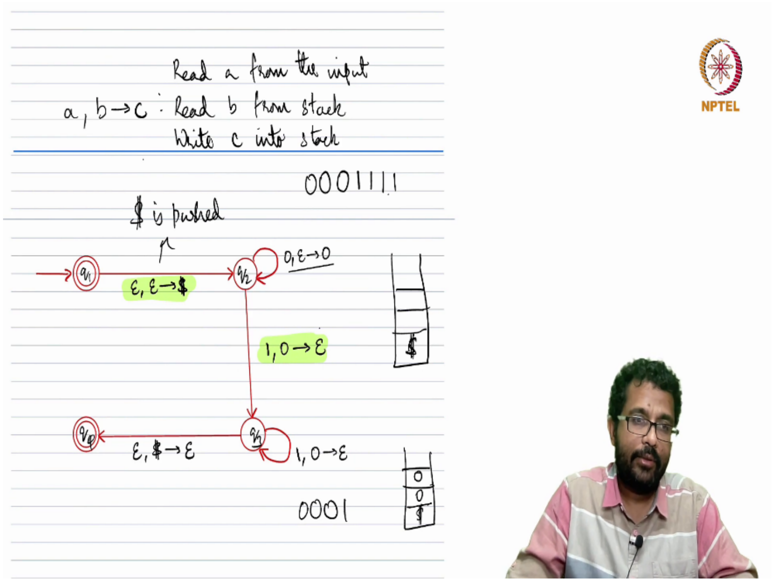
So, maybe when I say it like this, it seems a bit complex, but, when we see an example this will be clear. And like that, we make all these transitions finally, we reach let us say after you read the input, we reach the state rm, rm is the last state and for the string to be accepted this has to be a accepting state.

So, the only rule for accepting a string is that rm is an accepting state, this is exactly what we had in NFA also, but in the case of PDA pushdown automata we also have a stack. So, one may wonder whether do we want something with respect to the stack. So, one may think that if you have a lot of content in the stack, but at the end the stack is not empty, do we still accept the string? So, the answer is when we accept the string, the stack may not be emptied, it is completely fine.

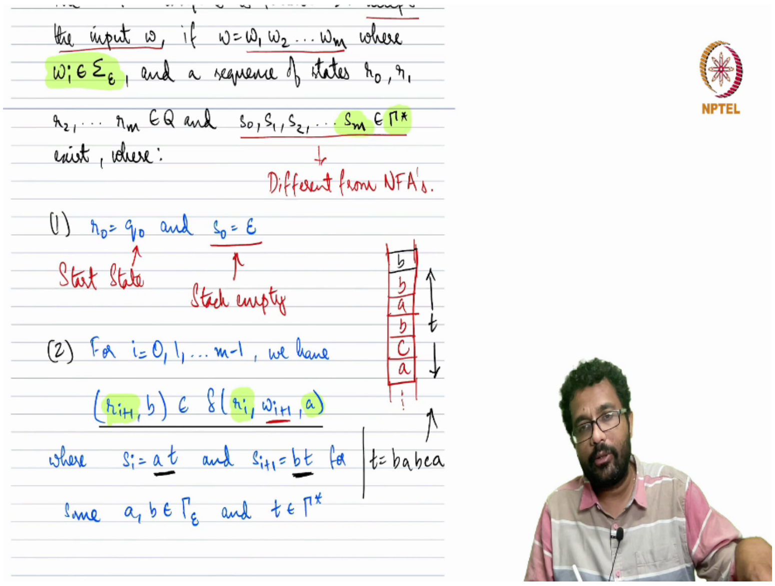
The only thing is that once you read the input string, we should be at an accepting state the stack need not be empty. In other words, in the sequence of stack, the strings that are in the stack, this sm this need not be an empty string for it to be accepted. So, these are the rules for accepting a certain string in the stack.

You should be decomposed in such a way that it starts correctly starting at the static state stack with empty and each rule should take be valid, each transition should be valid, ri plus 1 should be a valid successor of ri says that the si plus 1, the next stack configuration is a valid successor of the current stack configuration. And so, a and b should be such that and view read the correct input string. And finally, the last stage should be an accepting state

(Refer Slide Time: 24:13)



a, b→c : Read a from the input
Read b from stack
Write c into stack



the input w, if $w = w_1 w_2 \ldots w_m$ where $w_i \in \Sigma_\varepsilon$, and a sequence of states $r_0, r_1, r_2, \ldots r_m \in Q$ and $s_0, s_1, s_2, \ldots s_m \in \Gamma^*$ exist, where:

Different from NFA's.

(1) $r_0 = q_0$ and $s_0 = \varepsilon$

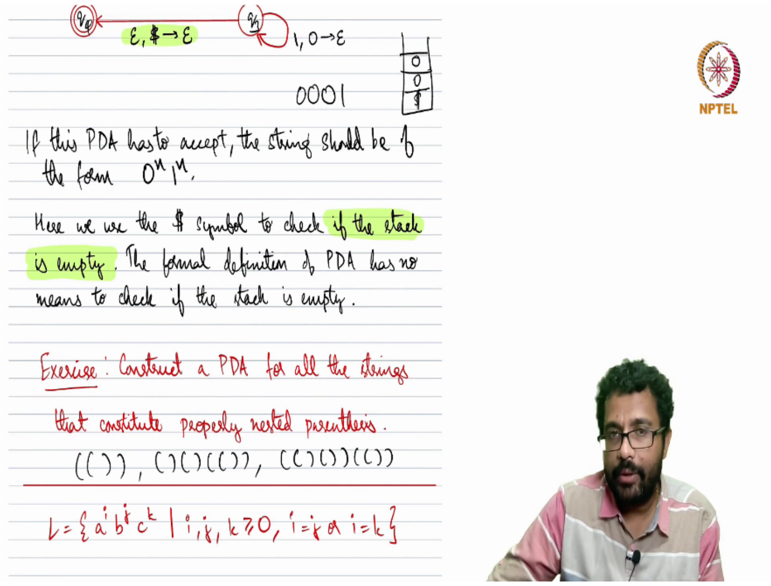
Start State

Stack empty

(2) For $i = 0, 1, \ldots m-1$, we have

$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$

where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$



If this PDA has to accept, the string should be of the form $0^n 1^n$.

Here we use the \$ symbol to check if the stack is empty. The formal definition of PDA has no means to check if the stack is empty.

Exercise: Construct a PDA for all the strings that constitute properly nested parenthesis.

(()), ()()(()), (()())(())

$L = \{a^i b^j c^k \mid i, j, k \geq 0, i = j \text{ or } i = k\}$

So now, let us just see couple of examples. So let us see this example. And let us see what this what this PDA does. So there are four states. So interestingly, the first state itself is an accepting state? And I have to explain this notation. So what this means, let us say let us take this notation, epsilon, epsilon, arrow, dollar, this means that the one before the comma epsilon, this means that maybe I will do one thing.

Maybe I will explain the, this one, because here are the three different symbols? So, this means that or maybe just right here a, b goes to c, this means that read a from the input. So when I say input, it is actual input, not the stack, and read b from stack. And write c into stack. So any of them could be empty, which means you could read nothing, you may read nothing from the input, you may read nothing from the stack, you may write nothing into the stack. So, all three also can be empty, in which case, it is just an epsilon transition.

So, one more point here, even over here? Even over here, the way I defined the transition, even over here, b could be empty, we may not write anything into the stack, a could be empty, we may not read anything into the stack, wi plus 1, which is the next symbol in the input that also could be empty, maybe all three can be empty. Which means without reading, or writing anything from to the stack or into the stack, or even not reading the input, you make the transition?

If both a and b are in empty, empty strings, that would mean that both si and sa plus 1 are the same. So you are not reading anything from the stack, you are not putting anything into the stack. So the stack remains intact? Anyway. So, a comma b, c, a comma b, arrow c means you read a from the input, you read b from the stack, when you read something from the stack, you are popping it. And then you write c into the stack, which means you are pushing it.

So what are we doing here, so first, this means the first transition, here it is basically $ is inserted. Nothing is read from the input nor the stack. So you first insert dollar into the stack. Maybe I will just use the word pushed instead of inserted. So if you are not familiar, you should be use the words push for inserting something to the stack and pop for removing something from the stack. So initially, dollar is pushed into the stack.

So $ is a special symbol that is in the stack alphabet. And then let us see what this loop does. So this means you are reading 0 from the input? And you are not writing anything. So you are not writing anything into the stack. Sorry, you are not reading anything from the stack. So, this is epsilon, but you are writing a 0 into the stack. So you can make go through this loops as long as you keep seeing 0 in the output. And whenever you see a 0, you do not read anything from the stack, but you write a 0 into the stack.

So it is like this, let us say the string is 00. Or it is, which is it starts with 000? So initially, without reading anything, you put dollar into the stack. So maybe I will just put it like this, you put dollar into the stack? And then you read a 0, and you put a 0 into the stack, then you read the second 0 and you put a 0 to the stack, then you put the third 0 and you put a 0 into the stack? Suppose that is it.

Now the next symbol is 1, let us say, so now you cannot remain in this loop. And the next transition to the next state is by reading a 1 from the input. So you are reading 1 from the input, which is what we have. And you are reading is 0 from the stack, you are reading 0 from the stack. This, but you are not writing anything into the stack, which means we are going to erase this top 0?

And now you come to the third state. I mean, let us say maybe we will just call it q1, q2, q3, and q4. You come to the third stage, let us say q3. Now, you cannot move to q4 because you cannot immediately move to q4 because q4 requires you to pop a dollar from the stack. So you are, you are reading a dollar from the stack, but right now you cannot read the dollar, the dollar is below the 0's. So if you want to pop the dollar, you should first pop the 0's.

The only way to pop the 0 is by using this loop on the q3, this particular loop. So if you want to pop a 0, you should read a 1. So now if you want to accept you should read ones but right now we read 0001 and we should have more 1. So if you have one more 1, you can pop it, you can pop one more 0 if you have one more, another one more, then you can pop both the 0. So, at this point, this is popped and this is also popped and now you can pop that out.

So now if you see, if the string was 000111, you would have popped the dollar and go gone to q4, which is an accepting state. But suppose you had 000111, and yet another 1. So, in which case after reading, let us say if you go to q4 by popping the dollar, then there are still there is still string left and there is no more moves left. Or if you were at q3, you were you had popped the two zeroes, you had popped the two zeroes.

You are at q3. And but there is one more 1, but you cannot even remain at q3 now because from q3 there are only two options one is remain at q3 or go to q4. If you go to q4, it is fine, you can go but this there is one more bit in the string length. If you try to read the one at q3 itself, that is also not possible because reading the one requires you to pop a 0 and there is no 0 in the stack.

So which means 0001111 is not going to be accepted. And similarly 0001 is also not going to be accepted because if you read this you will have a dollar and then 0 and then 0 like this and you cannot move to q4. So, in order to get to an acceptance, you need to move to q4. So, if you now if you are observing what is happening.

So, you initially push a dollar for every 0 read you push a 0 for every one read you pop a 0 and then finally, you need to remove the dollar as well. So, the only way to get to remove the find that the initial dollar that you pushed is whatever 0 you read same number of ones also should be read. So, the string should be of the form. So, the string if this PDA has to accept the string should be of the form 0^n1^n the same number of zeros and ones only then so whenever 0 is read you push a 0 and then whenever you read a 1 you pop that 0.

So, you also notice the significance of this dollar that is pushed and popped. Because if there is no dollar that is pushed at the beginning, there is no way to tell the stack has entered or no way to tell that you cannot read anymore. So this dollar is a special symbol that we were only pushing it once that tells you that severe read all the your pushed some zeros, you popped out the zeros.

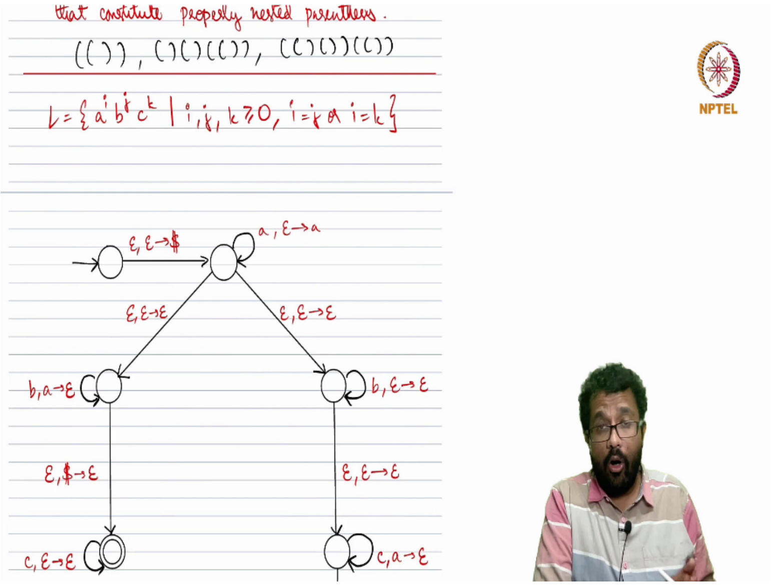
Now we are at the end whatever we have pushed we have popped this we need to know or we let us say we were pushed out the zeros, we are not popped all the zeros, the next symbol is still 0 which means the, the dollar is buried under many other zeros. So this, this dollar is an indicator that we have reached the bottom of the stack.

That is the significance of the symbol dollar. And that is what I have written here this dollar is used to check if the stack is actually empty. Because in the definition of the PDA itself, we said that upon acceptance, the stack may not be empty, that was not an essential thing. So this dollar is an artificial it is kind of a workaround to ensure that the stack gets emptied. So, this is a PDA for all these things 0 power n 1 power n.

So now already you see that this PDA is accepting a language which is not a regular language. In fact, it is a context free language. I think I mentioned this before, we will after in the couple of after one or two lectures will see that PDA's is the class of languages that PDA's recognizes is the same as context free languages. So we already know that 0 power n 1 power n is a context free language. So now we are seeing a PDA for it.

So one exercise is to construct a PDA for all the strings that constitute properly nested parenthesis. So, strings like this, all these strings are properly nested and matching parentheses. So try to construct a PDA for this. I think we have seen a context free grammar, but try to come up with the PDA as well.

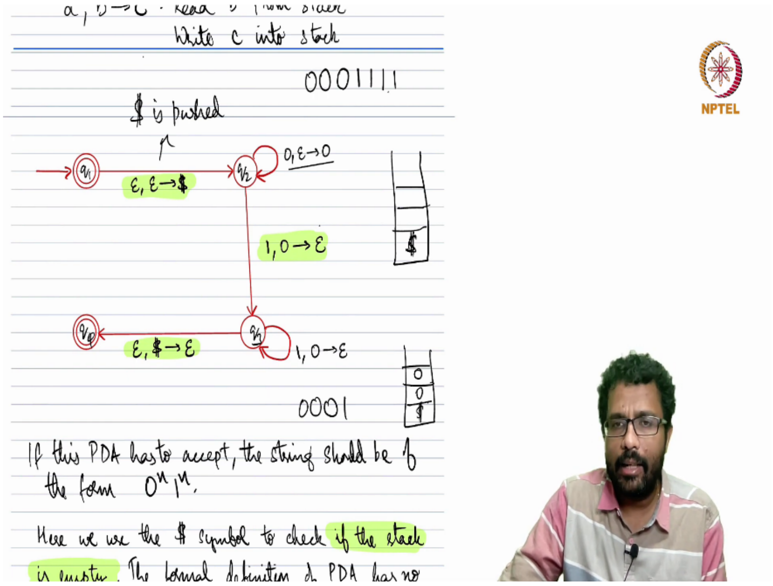
(Refer Slide Time: 35:44)



that constitute properly nested parentheses.

(()), ()()(()), (()())(())

$L = \{a^i b^j c^k \mid i, j, k \geq 0, i = j \text{ or } i = k\}$



a, b → c : Read b from stack
Write c into stack

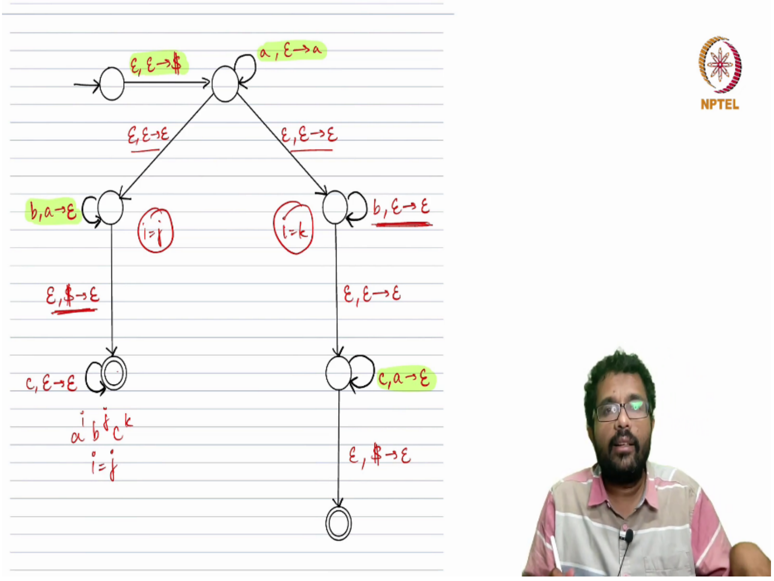
0001111

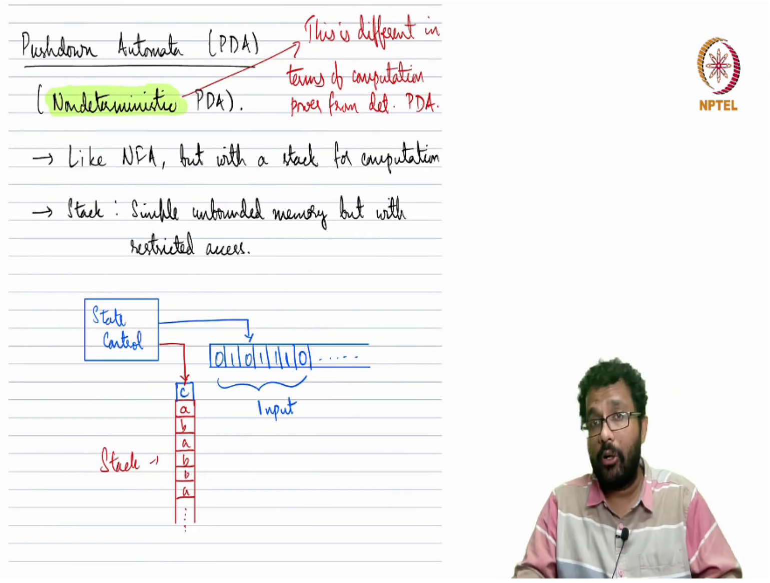
$ is pushed

0001

If this PDA has to accept, the string should be of the form $0^n 1^n$.

Here we use the $ symbol to check if the stack is empty. The formal definition of PDA has no





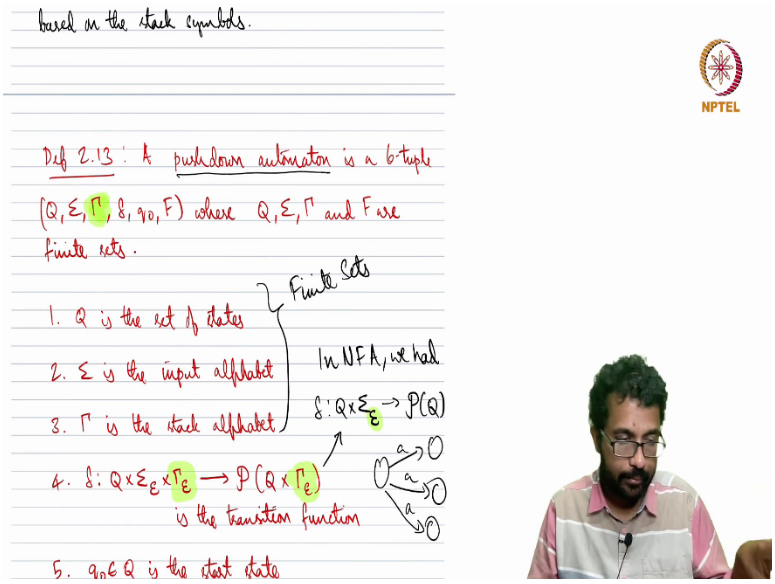
Pushdown Automata (PDA)

(Nondeterministic PDA).

This is different in terms of computation power from det. PDA.

→ Like NFA, but with a stack for computation

→ Stack : Simple unbounded memory but with restricted access.



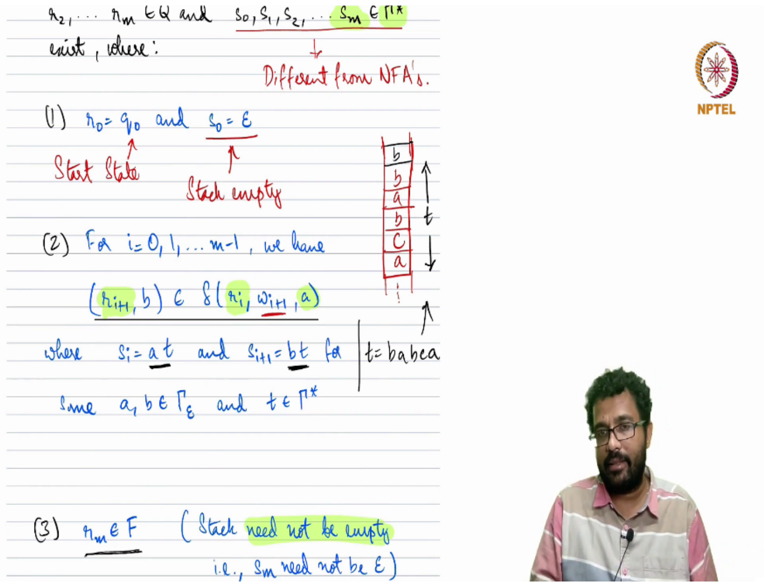
based on the stack symbols.

Def 2.13 : A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are finite sets.

1. Q is the set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
5. $q_0 \in Q$ is the start state

Finite Sets

In NFA, we had $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$



$r_2, \ldots r_m \in Q$ and $s_0, s_1, s_2, \ldots s_m \in \Gamma^*$
exist, where:

Different from NFA's.

(1) $r_0 = q_0$ and $s_0 = \varepsilon$

Start State

Stack empty

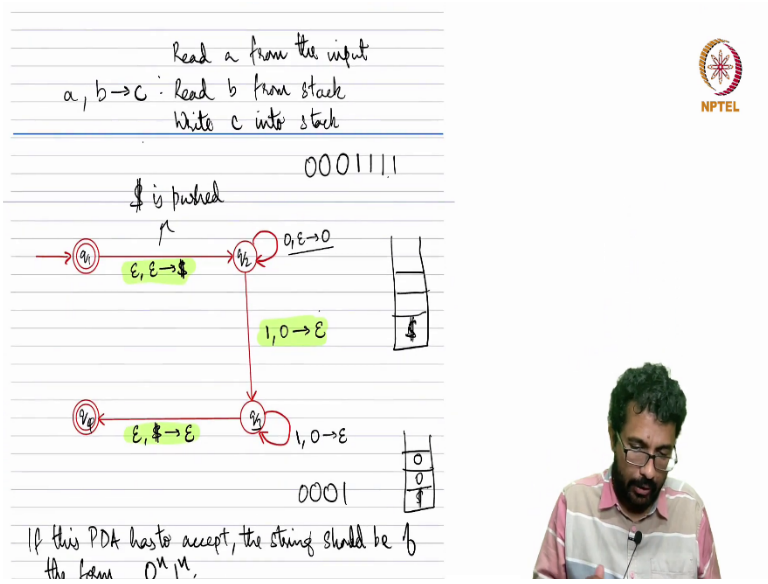
(2) For $i = 0, 1, \ldots m-1$, we have

$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$

where $s_i = at$ and $s_{i+1} = bt$ for

some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$

(3) $r_m \in F$ (Stack need not be empty i.e., s_m need not be ε)



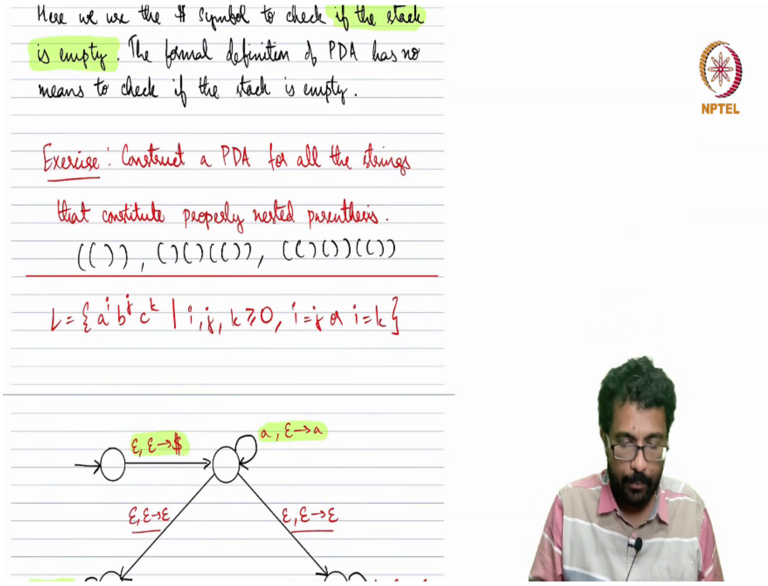
$a, b \to c$: Read a from the input
Read b from stack
Write c into stack

0001111

\$ is pushed

0001

If this PDA has to accept, the string should be of the form $0^n 1^n$.



Here we use the \$ symbol to check if the stack is empty. The formal definition of PDA has no means to check if the stack is empty.

Exercise: Construct a PDA for all the strings that constitute properly nested parenthesis.

(( )) , ( )( )(( )) , (( )( ))(( ))

$L = \{a^i b^j c^k \mid i, j, k \geq 0, i = j \text{ or } i = k\}$

$\varepsilon, \varepsilon \to \$$
$a, \varepsilon \to a$
$\varepsilon, \varepsilon \to \varepsilon$
$\varepsilon, \varepsilon \to \varepsilon$

So now we will see one more PDA, which is for the language a power i, b power j, c power k, where either i is equal to j or i equal to k. So either the number of A's is equal to number of B’s, or the number of A's is equal to the number of C's. So, it is something similar to what we did here, where here, we just had to check with the number of zeros equal to the ones. But here there is a choice to be made.

So, we use similar technique b, we use the stack to check that number of A’s is equal to B’s etcetera. But we do some, we have to use one more, one more trick. Before that, I just want to mention, so, we are really using the fact that the stack can be potentially unbounded, because 0 power n, 1 power n how much ever big one the n is we can push that many zeros into the stack because the stack does not have a bound.

So, that is why the stack has to be really unbounded. So, that is where we are using that. So, if you do not have an unbounded stack, then we cannot accept this language. So, suppose the stack is limited to size 100. In that case, we can only accept 0^{99} and 1^{99} because we have to put the dollar and the 99 zeros and then we are stuck. In which case it is a finite language, in which case it is not a context free language, it is just it is a context free language, but it is a regular language also, it is kind of trivial thing.

So, coming back a^k, b^k and $c^k$the number of A's equal to B’s are the number of A's equal to C's. So, we do the same thing here: we push a dollar at the beginning and we loop over here, we add an a to the stack for each symbol a that is read and then we make this we make this fork there are two possibilities. So in the left side, we check for, in the left side we are checking for i equal to j this side and this side it i equal to k.

So, the left side it is we are seeing whether the number of A's is equal to the number of B’s and the right side we are checking whether the number of A's is equal to the number of C's. And what is happening here this is both epsilon transitions. So this is an epsilon transition meaning you read nothing from the input, you read nothing from the stack, you write nothing into the stack.

So basically there are two it is just like to you basically take one of these two paths. In case of the left path, you see that for each b read, you are popping out an a. So, the only way that we will be able to see the dollar next. So, the so we will get to this the last state the accepting state in the left side, only if you are able to pop the dollar and that will happen only when the number of b’s read is equal to the number of a's and once you are able to pop the dollar you come to the accepting state and then you can accept.

So you will also accept let us say a power n b power n? Even if there are no c's, that is okay. But then once it is the number of b's and a's are equal. Now you can accept any number of c's as long as we only see c's. If we see a b here, we will not accept because there is no transition available for a b. But c's there are no limits. So, you can keep seeing c's, and you will still be in the same state. So, all the strings are the form a power n, b power n, c power n or maybe I will say a power i, b power j, c power k, where i equal to j will be accepted in the state.

The right side, it is similar basically, now, we have already read some number of a's and we have pushed that many same number of a's into the stack then, we are going to go into see these b’s and this there is nothing we are not we are seeing b’s, but we are not doing anything to the stack, we are just remaining in the same state, we cannot read a’s now, once we move this place, we cannot read c’s also. And at some point, we move to the next state.

And then we are reading c’s. And whenever we read one symbol c we pop out an a. So, that is what is happening here whenever we read a c we pop out an a. And the only way we will move to the accepting state is if the number of c's is equal to the number of a's. And the way in which these are ordered, it ensures that the string has to be of the form a star b star c star.

So, the only strings that we that that reached the last state here, the accepting state in the right side is of the form a power i, b power j, c power k. where i equal to k. So the number of b’s does not really matter because it does not touch the stack? That is what we saw here. So basically, it is the same, it is exactly the same thing that we did here that we did here. But here we are matching.

First thing is that we have a fork where we decide to check whether a is equal to b or a is equal to c. And the second thing is that when we are taking a equal to b, we have to kind of add the self-loop for c where it does not touch this stack. And when we are checking whether a is equal to c in the right side, we have to make the self-loop for b that b does not disturb the stack. So that is so this for this language a power i b power j c power k, i equal to j or i equal to k, we are able to construct a PDA. I think we are already kind of, this lecture has gone slightly over time.

So I think I will wind up with this part. So we saw water pushdown automata, which is actually non-deterministic pushdown automata, although we will not be repeating that anymore, it is like an NFA with a stack and the transition rules can be influenced by what is the stack content. And we explained when it exits, so basically it has to make valid moves and at the end it has to be an accepting state after having completely read the input.

So then we saw some example one was 0 power n, 1 power n and this two was a power i, b power j, c power k, where i equal to j or i equal to k. We also had this exercise for properly nested parenthesis that you can work out. And there are some other properties that we will see in the next lecture. But for now, I just I will stop now.