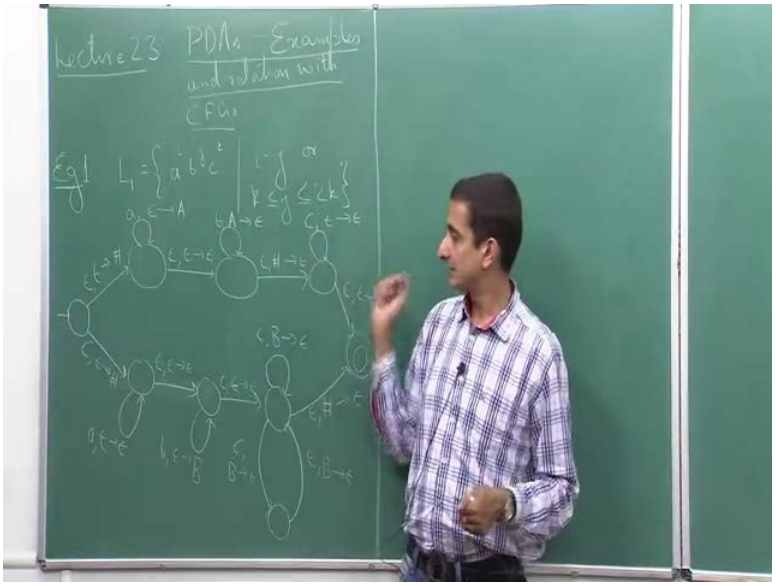
Theory of Computation
Prof. Raghunath Tewari
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture – 23
Pushdown Automata - Examples and Relation with CFGs

Hello everybody, welcome to lecture number 23 of this course. So, today, we are going to start by looking at some more examples of Pushdown Automata.

(Refer Slide Time: 00:18)



Then, we will try to explore the connection between context free grammars and push down automata. In other words, we will see that if I look at the class of languages accepted by push down automata versus the class of language is accepted by context free grammars, what are the relations between these two classes. So, we will let us start by looking at the following example.

So, our first example is going to be a language let us call it L 1. So, it has strings over the alphabet a b c of the form a to the power i b to the power j c to the power k, such that either i equals j, or j lies between k and 2 k. If either number of a's or b's are equal, we will accept this string; or if the number of b's is lies between k and 2 k, where k is the

number of c's even then we will accept such string.

So, how do we construct a PDF for this language? The idea is that, so because we have a or over here we will use epsilon transitions, we will use non-determinism in the form of epsilon transition to branch of at the very beginning. So, even before I start reading the first symbol of this string, I branch of into two parts into two branches, where one will check whether the number of a's is equal to the number of b's that is i equal to j or not. And in the other branch, I will check whether j lies between k and 2 k or not. So, I am going to non-deterministically decide to do one of the two things and branch of accordingly.

Let us see we have a start state. So, from my start state on without reading anything and without changing the stack, the only thing that I will do is that I will add a marker at the bottom of the stack. So, I will add a hash symbol at the bottom of the stack, and I will do the same thing here as well. So, I go to these two states. So in this state, I will check whether i is equal to j; and from this state I will check whether j lies between k and 2 k. So, how do I check whether i is equal to j. For every small a that I see that I will pop in A some symbol, I will push some symbol onto the stack; and for every b, I will pop it out; and in the end, I will compare.

So, every time I see a let say I push in A onto the stack then I move to another state without changing the stack. Here for every b that I see I will pop out an A. And now at the end of seeing all a's and b's if I have only the symbol hash on the stack, I know that i is equal to j. If I have hash on top of the stack, I will just remove it I know that i is equal to j. And on this state, I will just see an arbitrary number of c's. If I see a c, I do not do anything to the stack, it does not matter what I do.

And finally, at the end of seeing all the c's, I will just go to a accept state. So, this portion of my push down automata accepts languages of the form or strings of the form a to the power i b to the power j where i is equal to j and c to the power k.

Now I am going to accept strings of the form, where j lies between k and 2 k. So, first I am going to see an arbitrary number of a's because i is irrelevant here. So, I do

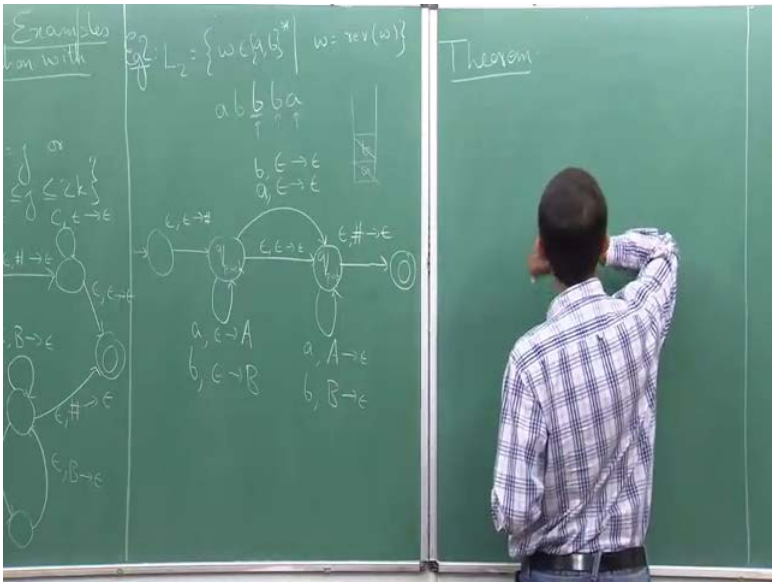
something like this. So, on seeing an ‘a’, I do not disturb the stack now I have to check this. So, what is the idea here how do we check whether the number of b’s is a between k and 2 k. The idea is we will again use non-determinism over here. What we will use that every time I encounter b, I will push a symbol on to the stack. So, finally, is equal to j on this stack, because that is the number of b’s.

Now when I encounter a c, either I will pop out a single b or I will pop out 2 b’s. So, for example, if the number of b’s is exactly twice the number of c’s that is j is equal to 2 k then for every c that I see I need to pop out 2 b’s; on the other hand, if k is equal to j then I need to pop out one. So, I will non-deterministically decide which 1 to do whether to pop of a single element from the stack on seeing a c or whether to pop of 2 elements from this stack on seeing a c, and this I keep on doing.

So, when I see a basically from here, I go to another state where first what do I is that on seeing b, I push in a symbol let say capital B on to the stack. Now I go to another state; and from this state, I will do one of two things. So, either on seeing a c, I pop out a B; or on seeing a single c, I will pop out 2 B’s. So, do that I will use another additional state. So, first on seeing a c, I will pop out a B which takes me to this state and from this state I will pop out another B without reading any symbol it is epsilon. These two operations together correspond to popping out 2 b’s on a single c, which takes me back to the state.

So, observe that I non-deterministically do one of the twp things in this state either this or this operation. And now finally, when all the c’s are popped out, I will just go to this state if the symbol on the stack is the hash symbol, I will pop it out and I will go to the accept state. So, this is the automata. Once again we are using non-determinism in two places. First, we used non-determinism to decide which of the two tests we are going to perform. Second, if we are going do the second check that is j lies between k and 2 k, we use non-determinism to either pop out a single B on saying a c or pop out 2 b’s on saying a c. These are the two things and the rest is pretty self explanatory.

(Refer Slide Time: 09:34)



Let us look at our second example. So, recall the definition of reverse of a string. So, we have this language L 2, which is also sometimes calling the palindrome. So, this consists of all strings w over let say a b star such that w equals reverse of w. So, strings which read the same no matter which side I read it, from left or from right. So, how do we accept string of this form using pushdown automata? The idea is we will again use non-determinism over here, and we will use non-determinism two kind of figure out, what is the middle point of the string. So, what we do is that I start scanning the string from the left hand side, and whenever I scan the string, I keep on pushing the symbols that I have read on to the stack.

For example, let say that I have a string a b a ab, let say abb and then we have b a. So, first what I do is that, so this is kind of the middle point of the string. So, till the middle point, I will keep on pushing symbols on to the stack, so first I push b. I kind of skip the middle point if it has odd length; and if it is even length, I will just push middle point as well. And then we compare with the rest for the remaining part of the string. If since I have b here, I will pop out this b; and then I have a here, so we will pop out this a, and I will accept. So, I will check the first part of this string with the second part of the string with the help of my stack. Now, I have to carefully handle the case when it is odd or when it is even - the length of the string.

So, here is the automata, so first on my start sate, I will push a symbol hash on to the stack, once again to check whether I have reach the end of the stack at any point or not. Now from this state, so this is the state, which will call respond to my push operation. If I see a, I will push a capital A on to the stack; and if I see a b, I will push a capital B on to the stack. So, it is on the same transition. If I see a, I will keep on pushing capital A; and if I see b, I will keep on pushing capital B. Now I have to go to a pops state. So, this state corresponds to, so let me call it q push and this will be q pop.

In this state, if I see a a, I will pop out an A; and if I see b, I will pop out a B. And now I check whether the top of the stack is hash or not. So, without reading anything, if hash is the symbol on the stack, I will just remove it and accept. And how do I go from q push to q pop. So, this is where I will use non-determinism. So, either the string has even length if the string has even length, then basically without reading any symbol; I go from q push to q pop even without changing the stack, because the first part and the second part are the same.

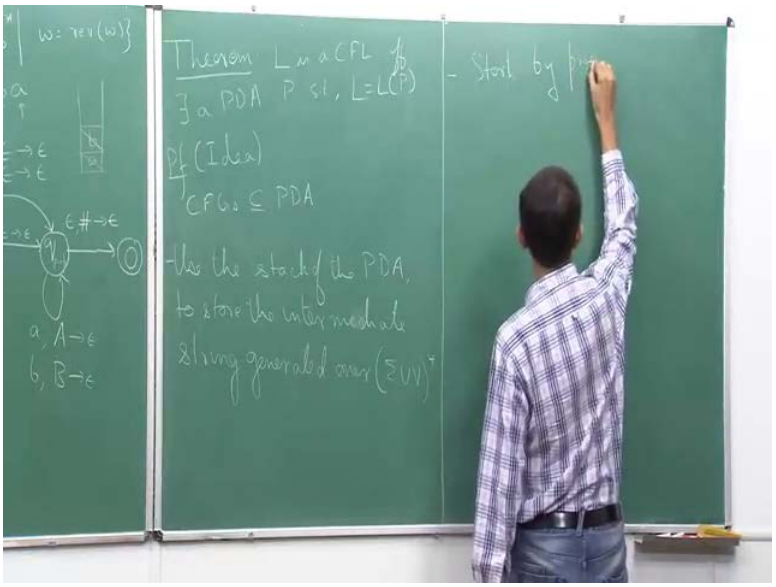
On the other hand, what can happen is may be the string has odd length, for example, the when this is the case if the string has odd length I do not change the stack for the middle symbol. So, when I get the middle symbol, I do not change the stack; I just read that symbol and proceed. If the middle symbol is a, I read a without changing the stack. Similarly, if the middle symbol is the b, I read b without changing the stack and this is the push down automata.

The crucial point is that at this state, we are making use of the fact that we are making use of non-determinism to find out what is the middle point of the string, because if I have reach the middle point then need to go to q pop. If I have not reached the middle point, I need to stay at q push that is what the automaton does in a nondeterministic manner. So, we have this example.

Now, we are going to look at the other aspect of today's lecture that is the relation between context free grammars and push down automata. So, how do these two computing objects compare in their power. So, as it turns out that these two computing objects they are equivalent in power in their expressive power.

Although they have a different structure one is a automaton structure, the other is a grammar, but the class of language is accepted by context free grammars is exactly equal to the class of languages accepted by push down automata and that is what to we are going to show in the next part. I am not going to go through the complete proof in detail, but I am going to give idea as to how the proof essentially works.

(Refer Slide Time: 16:24)



Let me state this as the theorem. So, L is context free language, if and only if there exist pushdown automaton P such that L equals L of P. So, recall that context free languages are exactly those languages, which are accepted by context free grammars. So, what we are saying is that if a language is a context free language that is if there is a context free grammar for a certain language then there is a push down automata which accepts it.

And if there is a push down automata that accepts the language, then it is the context free language which means that there is a grammar for it. So, we are going to see both the directions here. So, I am just going to write this as proof idea. Let us see why the first direction that is context free grammars or a subset of push down automata. Suppose, if we have a context free grammar, how do we create push down automata out of it.

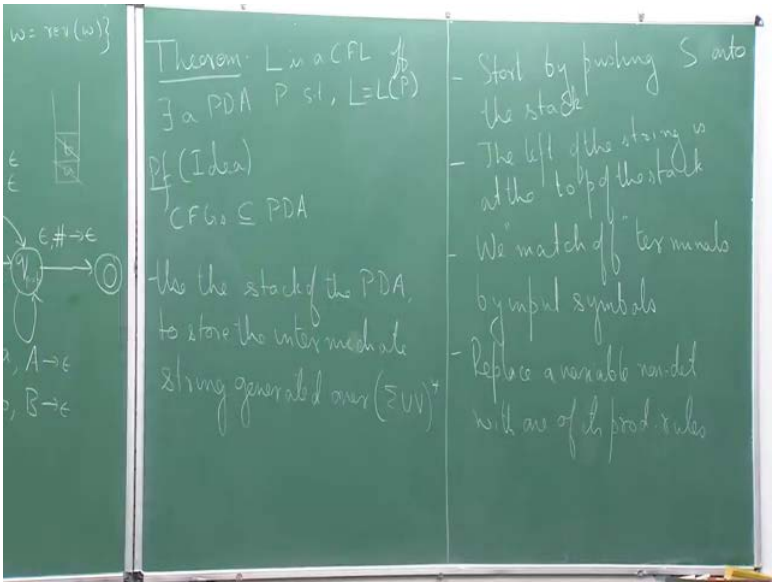
The idea is the following. So, what you do is that you take a context free grammar and

you use the stack to store the current string on the stack. So, we start with the start variable S, then we replace the start variable with some rule for S and then we get some other string over variables and terminals, then maybe I replace some other variable on the string to get another string, so these strings are called sentential string. So, I keep on getting this sequence of sentential strings.

If you recall the definition as to how a context free grammar accepts a string, it is basically a string is accepted by context free grammar. If there is sequence of this sentential string, these strings over terminals and variables such that one string can be derived from the previous string by replacing a single variable with it is production rule. And the last string is essentially just a string over terminals.

So, what we do is that we use the stack of the push down automata to remember all these sentential strings. And we keep on doing it. And whenever I have a, and basically the way I put it on the stack is that I put it in a top down fashion. The left portion of the string stays at the top of the stack. So, whenever I have terminals on this string, I will match them off with input symbols; and then when I have a variable, I non-deterministically choose some rule of the variable to replace that variable with that rule. Let me write it down to give a little better idea. So, use the stack of the push down automaton to store the intermediate strings generated over sigma union v star. So, as I said these strings are also called sentential strings.

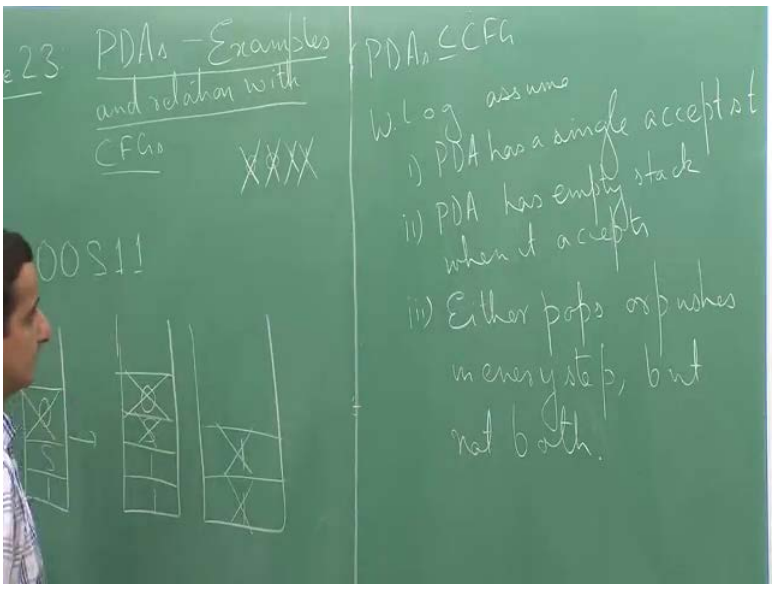
(Refer Slide Time: 21:29)



So, we use the stack of the PDA to store this, how do we start we start by pushing S onto the stack. So, S is our start variable, I push S onto the stack, and we do this. The left of the string is at the top of the stack, we match off terminals by input symbols. So, we create a transition function that will basically match off terminal symbols on the stack with input symbols and pop them out.

Basically match off means that if I have a terminal symbol, let say small a at the top of the stack, I will essentially read a small a from my input and pop of this small a from the stack that is what it means to match off. And we replace a variable non-deterministically with one of it is production rules. So, to motivate this, let us just look at a quick example. So, I am not going to give the entire example, but just a part of it.

(Refer Slide Time: 24:30)



Suppose we are looking at the grammar for 0 to the power n and 1 to the power n, so we have the rule S going to 0 S 1, and S going to epsilon. So, what we do is that, so essentially if I have let say I have generated at some point a string of the form 0 0 S 1 1. So, from S, I first generate 0 S 1 then I generate this. So, at this point, basically my stack, so what does it contains. So, initially I star with S on the stack, now since the top variable of the stack is a variable seen that top symbol of the stack is a variable, I replace S with it is production rule 0 S 1. So, I have 0, I have S and I have 1, I push these three symbols onto the stack.

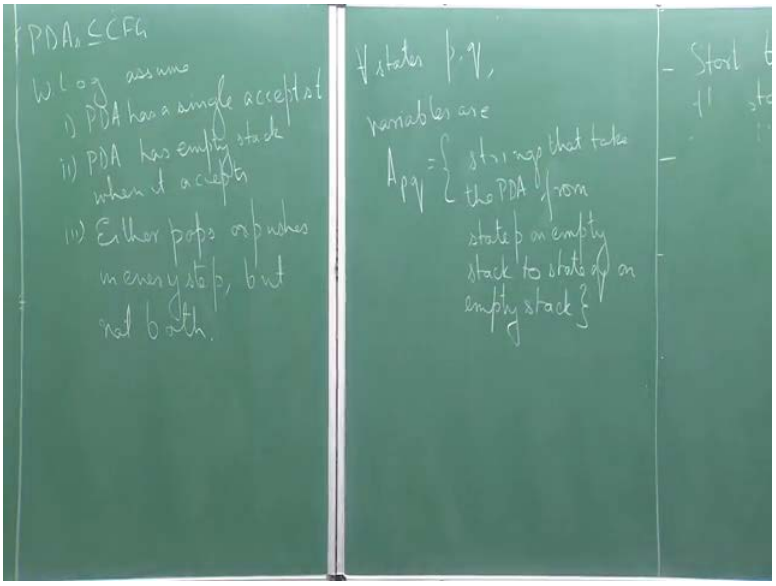
Now, I try to find what is the top most variable on the stack and to do this I will match off strings basically. Let say my input is something like 0 0 1 1. So, I match off this 0 with this 0. Now I have a variable on the top of my stack. So, I replace S with its production rule 0 0 S 1. So, I have 1 here, and I replace S with 0 S 1, so I get 0 S 1. Once again match off this 0 with this 0, again I have a variable at the top of my stack. So, I will replace S with its production rule epsilon, and therefore, I get 1 and 1 on the stack. Now I match off 1 with 1, and I match off this 1 with this 1. Now, the input is completely read and the stack has again become empty, so this gets accepted, so that is how it works.

For the other direction, we have to show that PDA‘s contain context free grammar. So,

here without loss of generality we assume the following. So, first we assume that the PDA has a single accept state. We next assume that PDA has empty stack when it accepts. And thirdly, we assume that it either pops or pushes in every step, but not both. So, this is the exclusive r. So, how can we ensure this, if the PDA has the multiple accept states, I just put epsilon transition from all these accept state to one accept state.

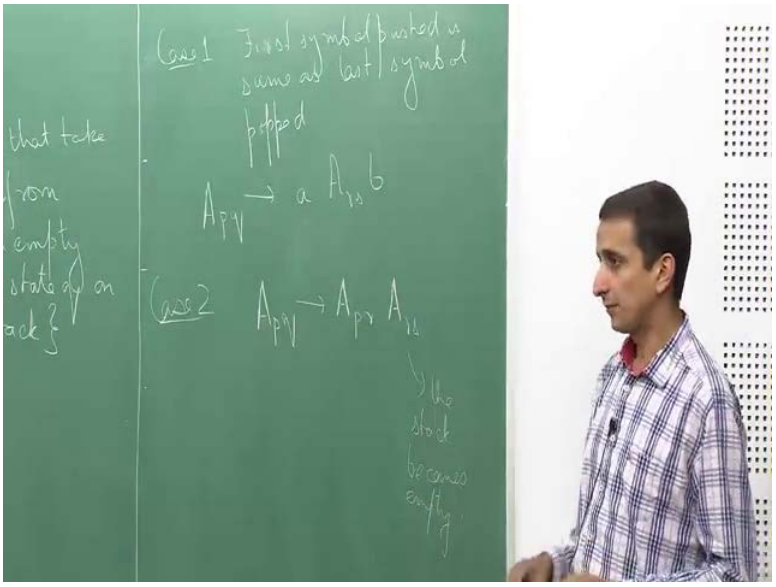
If the PDA when it accepts it does not empty out its stack, what we do is that if it reaches the accept state and it has exhausted the input, we will create a new accept state. And before going to the new accept state we will pop out everything out of this stack, we will make sure it is empty and then non-deterministically move to the new accept state once it has become empty. To ensure that it either pushes or pops in a every step, but not both. Suppose, it pushes as well as pops we just divide this transition into two separate transition where in one step it pushes, and in the next step it pops or vice versa.

(Refer Slide Time: 29:13)



Now that we have these conditions what we do is that we create variables of the form. So, for all p comma q of the PDA create variables of the form A p q. So, A p q will accept every string, so sorry A p q will generate all strings So, I will just write it as A p q is a set of all strings that take the PDA from state p on empty stack to state q on empty stack. So, once we have this, we will just now need to divide into two cases.

(Refer Slide Time: 31:02)



So, case 1 is, so let say from going from state p to state q, first symbol pushed is same as last symbol popped on the stack. If I look at the stack, the first symbol popped by a pushed is the same as the last symbol pop. So, in this case, we add a rule of the form A p q goes to small a R r s b, where this is the first input symbol that is read from going from p this is the next state that we go to when going from p to the next state. S is the state from where we go to q and b is the last input symbol that is read when going from S to q, this is case 1.

And case 2 is when this does not happen. The first symbol pushed is not the same as the last symbol read. There is some other state in between when this is the first symbol pushed gets popped out. So, in this case, we will add the rule A p q goes to A p r concatenated with A r s where r is the state where the stack becomes empty after p. So, this is the position where it becomes empty, because it needs to become empty at some point because the two symbols are not the same.

I will stop here, and we will continue next time.

Thank you.