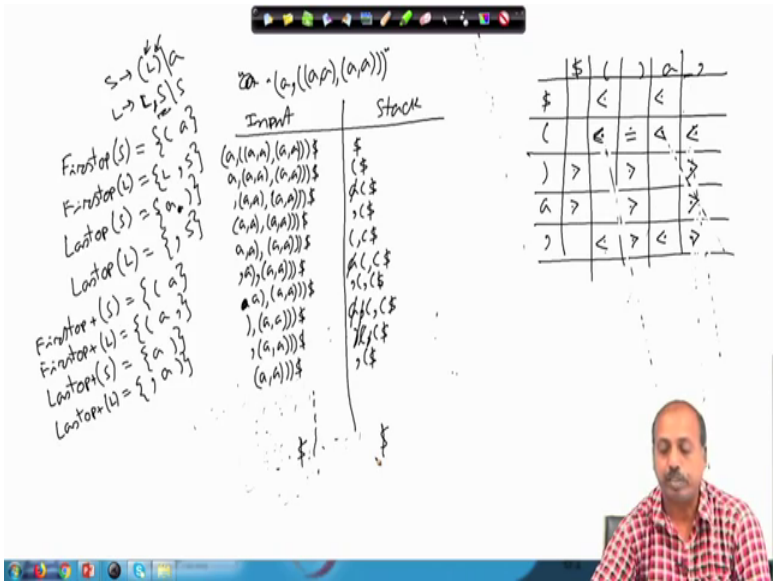
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 27
Parser (Contd.)

(Refer Slide Time: 00:15)



So, next we will be looking how into an example of how to use this particular table for doing a parsing operation. So, let us consider an input string which is given by this expression. So, this a, then sorry a, then comma then a comma a comma, then a comma a, then two close parenthesis. So, this is the string. So, we will see whether we can derive this string using parse this string using this operator precedence parsing table.

So, as you know that we will have two parts in our formulation; the first part will be the input and the stack. So, initially input is this whole string. So, this is a comma within bracket a comma a comma a comma a and it is terminated by dollar and the top of the stack is also dollar. So, we have got open parenthesis and dollar. So, top of the stack is dollar. So, dollar is less than open parenthesis. So, it will tell that it tells that I have to shift this open parenthesis into the stack. So, the configuration changes to something like this. Then, open parenthesis and dollar and then, this next input symbol is a and top of the stack is open parenthesis.

So, open parenthesis is less than a; open parenthesis is less than a. So, this will also be shifted into the stack. So, this input now becomes comma a comma a comma a comma a dollar and this becomes a open bracket dollar. Now by this rule, so comma and a; so, comma so top of the stack is a and a is of higher precedence than comma. So, a will be popped out from the stack. Now, last top of the stack is now open parenthesis and a is the symbol popped out. So, open parenthesis is open parenthesis is of lower precedence than this one, open parenthesis lower precedence than symbol a.

So, the popping process stops. So, it is at this situation. Now, we have got comma and open parenthesis and comma is of lower precedence than open parenthesis. So, it will be shifted. So, you will be the input string will be like this. So, it will be comma, open parenthesis and dollar. Now, we have got this open parenthesis. So, comma and open parenthesis, then comma is of lower precedence than open parenthesis. So, it will be shifted. So, the input will be like this.

So, open parenthesis will be shifted, then comma then another open parenthesis and dollar. So, this will be the situation. Now, open parenthesis and a; so open parenthesis and a, open parenthesis is of lower precedence than a. So, a will also be shifted. So, the situation will become, so, a will be shifted. So, a open parenthesis comma open parenthesis dollar. So, that is the stack. Now, between a and comma, a is of higher precedence than comma. So, a is popped out.

Now, top of the stack is open parenthesis and open parenthesis is of lower precedence than a. So, the popping process stops. Now you have got the situation; a now we have got the situation where comma is the next input symbol and open parenthesis is the top of the stack and this open parenthesis is of lower precedence than comma. So, comma will be shifted. So, we have got this situation comma is shifted fine. Next I have got comma and a and comma is of lower precedence than a. So, a will be shifted; a will be shifted ok.

Now, I have got a and closed parenthesis. So, a and closed parenthesis; a is of higher precedence than closed parenthesis. So, a is popped out from the stack. Now comma and a, so, comma and a; comma is of lower precedence than a. So, this pop process stops. Now, I have got this closed parenthesis and comma. So, closed parenthesis is of higher precedence sorry comma and closed parenthesis. So, comma is of higher precedence than

closed parenthesis. So, comma will be taken out from the stack and now I have got open parenthesis and comma.

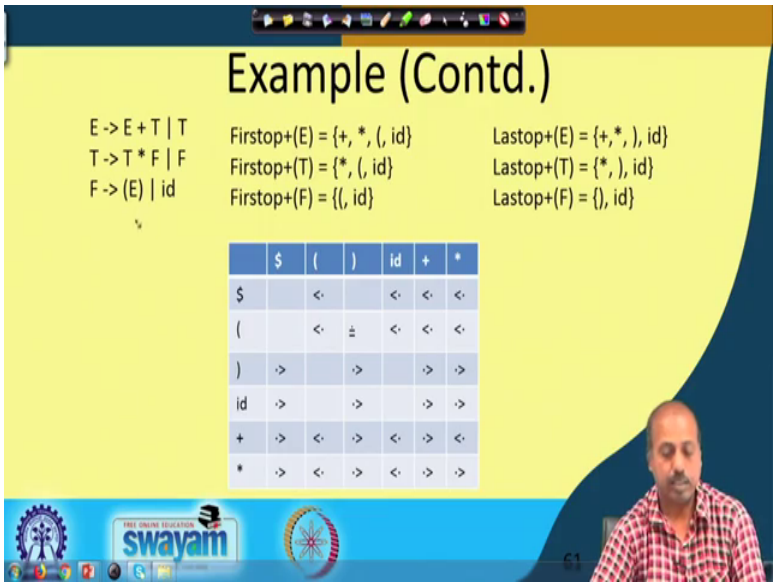
So, open parenthesis is of lower precedence than comma. So, the popping process stops. Now, I have got this closed parenthesis and open parenthesis. So, closed parenthesis is of higher precedence than open sorry, open parenthesis and closed parenthesis. So, open parenthesis and closed parenthesis, they are of equal precedence. So, they are so, it will be pushed into the pushed into the stack. So, this becomes comma a comma a; then this dollar, now I have got this open closed parenthesis open parenthesis comma open parenthesis dollar on to the stack.

Now, top of the stack contains closed parenthesis and the next input symbol is comma. So, closed parenthesis comma higher precedence. So, this is popped out. Now top of the stack contains open parenthesis and last symbol popped out is closed parenthesis. So, open parenthesis is of equal precedence as closed parenthesis. So, this is also popped out. Now top of the stack contains comma and symbol popped out is closed parenthesis. So, closed comma is of higher precedence than closed parenthesis. So, this comma will also be popped out. This, comma is also popped out.

Now, you have got this plus and dollar. So, now, this open parenthesis, this open parenthesis and comma. So, open parenthesis and comma. So, open parenthesis of lower precedence than comma. So, the process stops. So, we have got the next situation as open parenthesis and comma. So, open parenthesis and comma say it should be less than. So, it will go to a comma a, then dollar then this would be comma open parenthesis dollar.

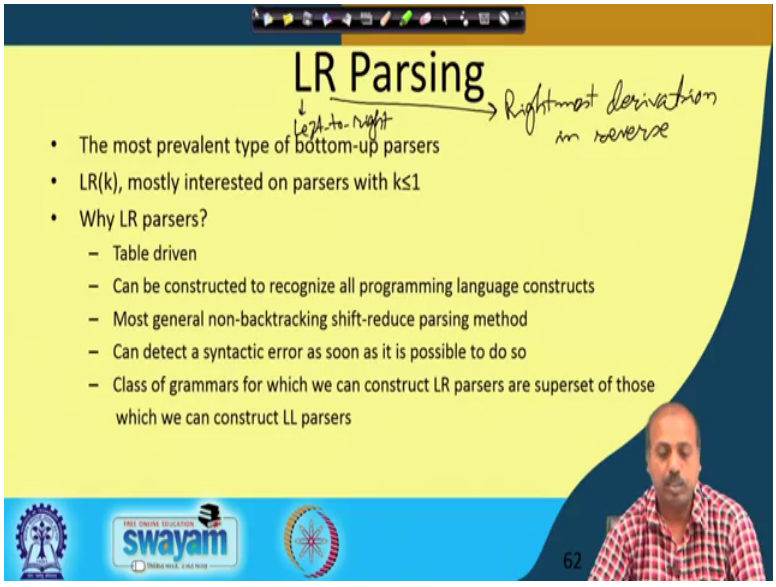
So, it will proceed like this ok. So, this way you can continue in this table. So, ultimately we will find that the top of the stack input will also be a dollar and top of the stack will also be dollar. In that case it will be in an accept state. So, you can just continue few more steps to get the final table.

(Refer Slide Time: 08:49)



Next we will be looking into the LR parsing process which is even a better version than operator precedence because operator precedence parsing the basic difficulty that we had is that it cannot give us for parser for many of the grammars which do not have this operator grammars structure ok.

(Refer Slide Time: 09:03)



So, this LR parser. So, this is the most prevalent type of bottom up parser and LR k is mostly; so, in general we will call LR k. So, where, k is the number of symbols that we do a look ahead. So, this LR; so, this L and R, so these two letters L stands for L stands

for this left to right scan; L stands for left to right scan and then this R stands for rightmost derivation in the reverse; rightmost derivation in reverse. So, it is in reverse because it is a bottom up approach. So, it is that is why it is in reverse.

So, it produces rightmost derivation unlike that LL, so, which was producing a left most derivation. So, this will give a right most derivation. Both LL and LR they will do a left to right scan, but the which symbol to be replaced, so that will differ. So, we in general we will have got LR k parser. So, where k is the number of tokens that you do a look ahead or number of symbols that you look ahead. Now, in general we will have k value less or equal 1. So, will be looking into LR 0 and LR 1 grammars. So, will be looking into LR 0 and LR 1 grammars.

So, LR 0 will give us the SLR parsing table; whereas, this LR 1 will give us the canonical LR type of structure. Now, why should we go for this LR parser because most of the parsers that we will find, so they will be of this nature. They will be LR parser. So, why should we have this? So, first of all this is a table driven parsing. So, it is there is no recursion involved in it and can be constructed to recognize almost all programming language constructs. So, here though it written here as all, but it is it need not be also it is almost all.

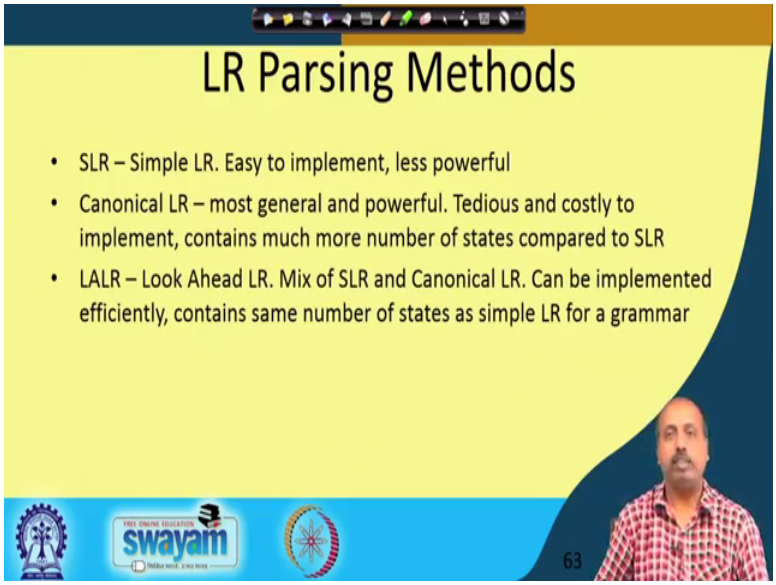
So, and it is most general non backtracking shift reduce parsing method. So, it if there is no backtracking involved, so, you do this coding for this is pretty easy and there is no it follows the shift reduce parsing policy that is fine. So, can detect a syntactic error as soon as it is possible to do so. So, this is the fastest method to detect syntactical errors. Main reason is that it is following a bottom up approach and it is trying to identify the handles in the stack.

So, that is why as soon as it finds that there is possible handle, but it cannot be derived from any of the inputs any of the sentential forms; in that case it can flash the error. So, this way it can find out syntactic errors as fast as possible and this is the most important observation that the class of grammars for which we can construct LR parsers are the super set of those for which we can construct LL parsers.

So, if for a language you can construct LL parser, so you can also construct LR parser for that. But the reverse may not be true. So, you may be having only the you may not be

able to formulate LL parser, if we have got if we for language that can support you getting LR parser.

(Refer Slide Time: 12:47)



So, there are three different methods of this LR parsing that will be discussing in this course. One is the SLR parsing or simple LR, they are easy to implement and it is less powerful. So, powerful in the sense that the class of languages for which you can construct a parser without any shift-reduce conflict or reduce-reduce conflict.

So, without any conflict so the set of languages for which you can construct a parser. So, if that set is pretty large, we will say that the parsing strategy is powerful. If it is not that, so we will say it is less powerful. Then we have got Canonical LR, it is a most general and powerful. So, this; so, this is the I should say the most general so for whatever language you can construct a shift reduce parser. So, we can do a we can construct a canonical LR parser for that. However, the difficulty is it is tedious. So, you will see that there are large number of states that will be created in the parser. So, that way it is difficult to make to make a CLR or Canonical LR parser like that.

And costly to implement because since the number of states are more, so you will have this policy of the program that will be rise that will be used for getting this parser will be difficult. So, contains much more number of states compared to the SLR parser. So, when we look into some example, it will be more clear. Then another class of parser

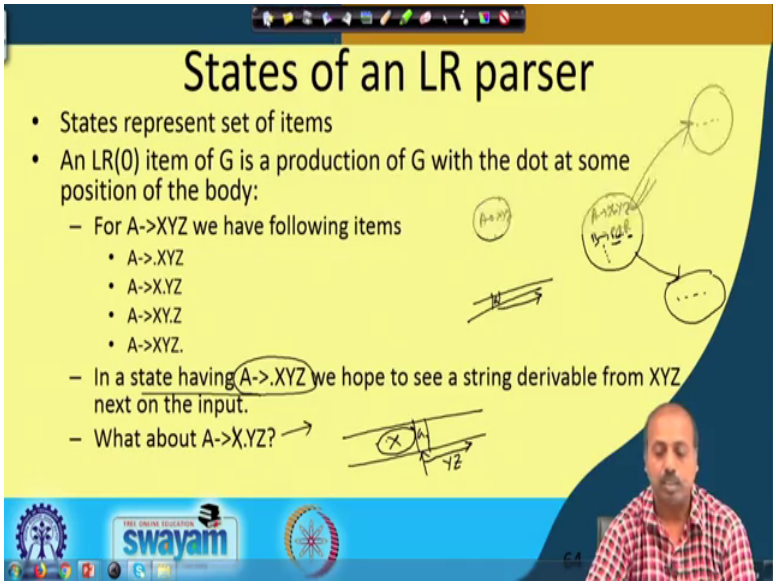
which is known as LALR which is look ahead LR. So, it will do some look ahead and try to see whether it can do better in terms of resolving the shift reduce conflicts.

It is a mix of SLR and canonical LR can be implemented efficiently and most importantly it contains same number of states as simple LR for a grammar. So, it is it has got same number of states as your SLR parser, but power wise it is same as the Canonical LR parser. So, what we normally do is that we construct the Canonical LR parser and from there by doing some equivalency analysis. So, we try to merge the states and come reduce that Canonical LR parser to a SLR to an to an LALR parser and the resulting LALR parser will have same number of states as the SLR parser

However it will be much more powerful than the SLR parser and its power will be same as that of the canonical LR parser. So, that is why most of the automated tools that we have. So, they will try to construct an LALR parser though the construction process is difficult. So, if you if you are trying by hand you using a pen and pencil, pencil and paper type of approach.

So, doing it by hand, then you can most of the time. So, you will find that you can make the SLR parser by hand, but doing the canonical LR or the LALR parser by hand is a very tedious job. So, excepting for very simple grammars, so will not be able to do it by hand.

(Refer Slide Time: 16:01)



So, all this LR parsers, so you have got a concept of State and a state represent a set of items. So, what is an item that we will define and LR 0 item. So, when I say LR 0; that means, it does not do any look ahead's. So, it just looks into the current input symbol and based on the state in which the parser is and the next current input symbol, it will take a decision like what to do. So, what which state to go next or what should be the action.

As we know that there all are four action shift, reduce, accept and error. So, which action to be taken? So, it will be deciding on that. So, if there is a production a producing XYZ. So, like this then we can have. So, all these are items. So, an item you can say that it is like a production rule with a dot somewhere on the right hand side. So, here, so this is an item where dot is at the beginning; this is an item where dot is after the X symbol. So, this is after the Y symbol and this is after the Z symbol.

So, that is the notation, but what do we really mean by that? So, when we say that we are in a state that has got this as an item. So, we are in a state where we have got this a producing dot say dot XYZ as an item; that means, in this state I am expecting to see a string which is derivable from XYZ on the next input. So, whatever be the next input symbol say A, so that means, if you if I am currently in this state and the next input symbol is A. Then it will be it will be leading to derive a string. So, from this point I will be I will expect that I will find a string which is derivable from XYZ.

So, this way it will try to do a prediction like it will try to a figure out which rule which type of derivation I am going to see and that is what it will be doing. Now what about this one? So, Y producing X dot YZ. So, this means we have already seen a part of the string we have already see seen a substring which is derivable from X. So, as if this is the next if this is the input, then we are at this point a. Then, this part in this region I have already seen a string which is derivable from X and then, the next input symbol is a and from this point I am expecting to see a string which is derivable from YZ ok.

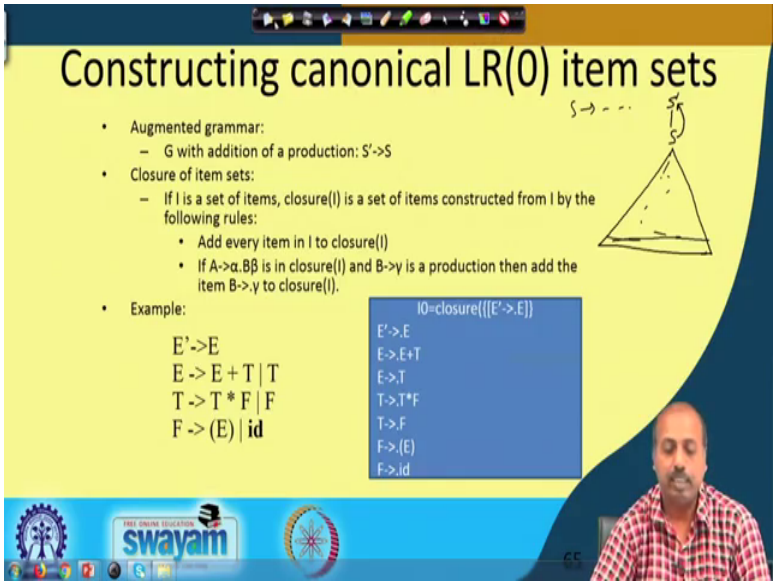
So, that way, so this dot is very important. So, dot means the portion before dot is I have already seen that. So, in some state, so if I have; so if I have what say number of items like say A producing X dot YZ or say B producing say PQ dot R like that. So, if I have got say two items that means, I could have I have arrived to this state either by either of these two cases; like I have seen a string which is which is derivable from X and now I

am expecting to see a string which is derivable from YZ or I have seen a string which is derivable from PQ and now I am expecting to see a string which is derivable from R.

So, this way, so all the items that you have in a state. So, that they will tell us that they will give us enough hint about the type of string that we have seen so far and the type of string that we are going to we are expecting next to see. So, based on that we have to take a decision, like at this point; so if the next input is such that so this one survives. So, X producing YZ survives. So, will be going to another state where it will be it will be advancing input and it will expect some string that way.

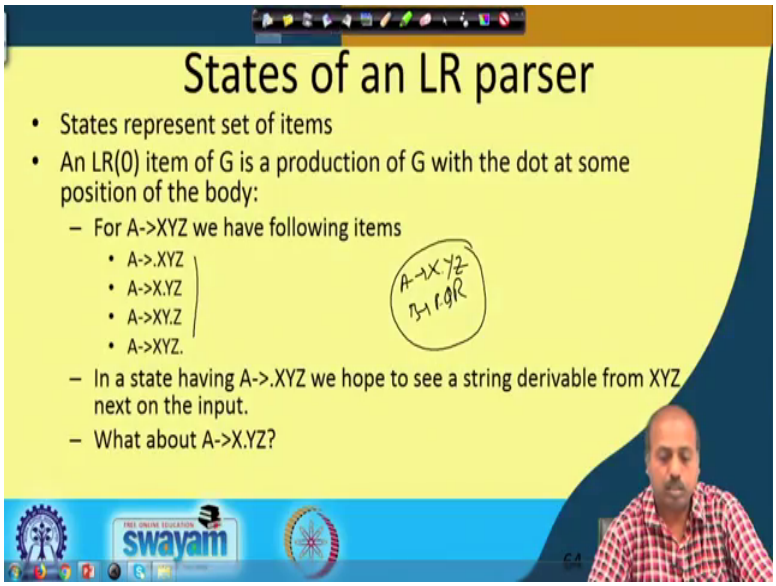
On the other hand, if the next input is such that so this alternative survives ok. So, this is the this is not in that case, it will be coming to a state where it will be doing like this. Now if the parser cannot take any decision like, then of course, there will be an error. So, shift-reduce conflict or reduce-reduce conflict can occur. So, we will see that as we proceed through this slides. So, this is the meaning of A producing X dot YZ and that is we have already seen is a string derivable from X and then, we are expecting to see a string which is derivable from YZ.

(Refer Slide Time: 20:39)



Now, how do we construct this LR 0 items ok. So, canonical LR 0 items, item sets, because it is not item. So, item sets because each of them each of this thing like A producing dot something. So, that is a, that is an item as we have seen in the last slide. So, the all of these are items. So, all of these are items.

(Refer Slide Time: 21:01)



So, when we say a set of items, so, this is basically a collection of items that we had like A producing X dot YZ; so, B producing P dot QR. So, these are all; so, the individually they are item. So, this collection of items so that is called set of items and that will constitute a state.

So, we see that um. So, how to construct this LR 0 item sets? So, first thing that we have to do is to augment the grammar by another rule S dash producing S ok. So, why do we do this thing? So, previously we have a, we had a grammar that started with S and all these rules were there. So, in the ultimate production or that parse tree that is produced. So, at the root we had got S and then from here everything is derived. So, this is there are lowest levels. So, we have got all the terminal symbols. So, what we are doing? So, we are adding another rule is just producing S.

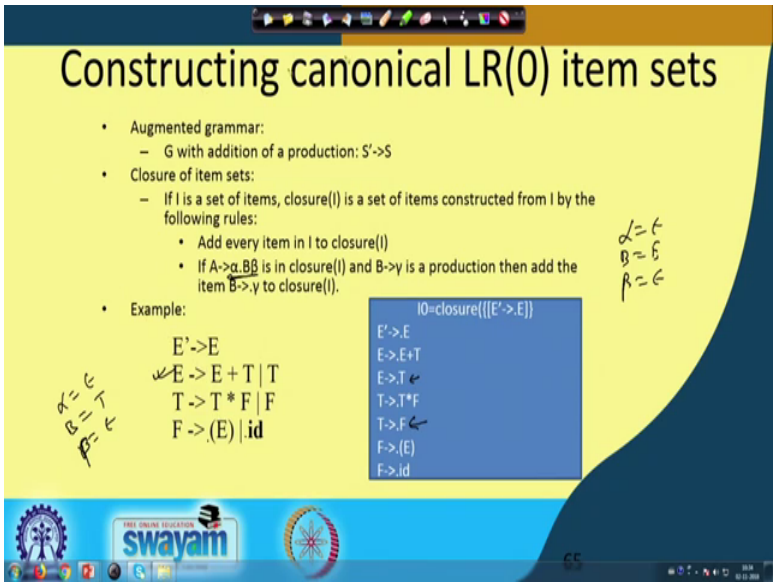
So, why do we do this? We do this because when the parser will try to do this reduction S to S dash; then, we know that we have seen the we see we have been successfully parse the whole string. So, that is the idea of adding this extra production is just producing S into this grammar set G and call it an Augmented grammar. So, we augment the grammar G by adding the rule is just producing S; then we construct the closure of item sets. So, if I is a set of items, closure I is a set of items constructed from I by this rules.

So, at every item of I to closure of I and if A producing alpha dot B beta is in closure of I and B producing gamma is a production, then add B producing dot gamma to the closure

of I. So, this is how will be doing the closure. So, suppose we have got a grammar like this E dash producing; so, this is that augmented ETF grammar. So, we have got this original grammar had these three rules E T and F and we have added another rule E dash producing E add as the extra rule.

Now, say I 0 that is items item set 0. So, this is this was having the only one rule E dash producing dot E. So, this is for the whole graph this is for the situation where I have not yet seen anything. So, I am expecting to see a string which is derivable from E. So, E dash producing dot E. So, when I take closure of this, so by applying this rule. So, you see we have got E dash producing dot E. So, if you compare with this.

(Refer Slide Time: 24:07)



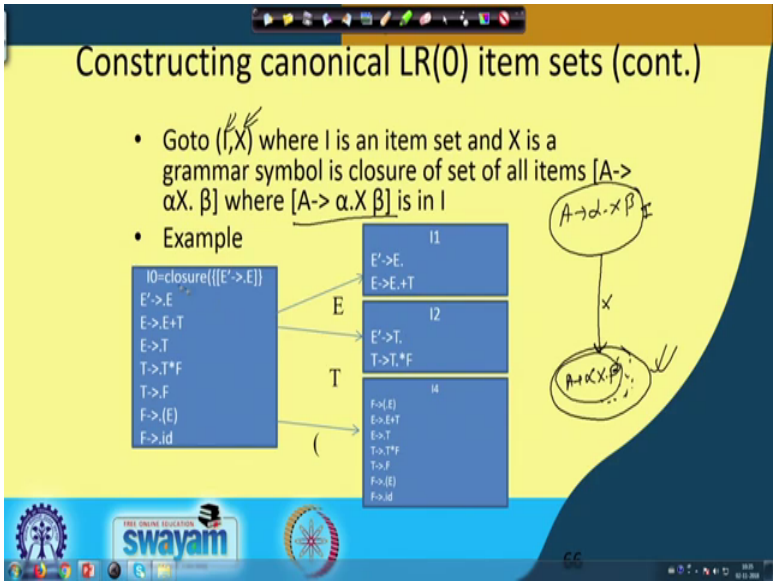
Then, alpha is epsilon; then B is E; B is equal to E and beta is also epsilon.

So, comparing with this, now it says that whatever we have so B producing gamma; so, B producing gamma; so, this rule survive. So, E producing E plus T and. So, if I do that, then it says that B producing dot gamma should be added to closure of I. So, E producing dot E plus T should be added to the set. So, E producing dot E plus T is added, then this E producing T, this also survives by the B producing gamma as per for this rule. So, E producing dot T is also added. Now, as soon as E producing dot T is there ok, Then again so this rule has to apply.

So, here alpha equal to epsilon B equal to T and gamma equal to epsilon and then, sorry beta equal to epsilon; beta equal to epsilon and then it says that B producing dot gamma. So, T producing dot T star F and T producing dot F. So, these two are added to the set and as soon as this F T producing dot F is added. So, based on that this F producing dot within bracket E and E at the F producing dot id. So, they are also added to the set. So, this way I can compute the closure of this particular set E dash producing dot E and that will constitute one state of items.

So, I 0 is the initial state of the parser where it starts with E dash producing dot E and takes the closure of that particular item to get the whole all the items in that set. So, in his way, so we can construct the items.

(Refer Slide Time: 26:05)



The another part of is to construct the Goto. So, Goto I, X. So, this is defined for an item I for an item I. So, on some grammar symbol X, we can define a Goto. So, if I is an item set and X is a grammar symbol is a closure of all the items a producing aX dot beta, where a producing alpha dot X beta is in I. So, it is basically since a producing alpha dot X beta is in the item. So, from this if I get X, then what we are planning to do so that is the thing.

So, if you if you get an X naturally you will be coming to a state where a producing alpha X dot beta will be an item because if you have seen X, then you will be expecting to see a string which is derivable from beta. So, this will be calling Goto I, X. So, the

Goto. So, this is the set I; this is X. So, Goto I, X is this set and you take the closure of this. So, to get all the items in this state. So, we have to take the closure of that one. So, this is an example like say this I 0 is the state which is I 0 is the state which is E dash producing dot E and the closure of that. So, this is the set of items that we have.

Now, from here on E, so from this rule you see the E dash producing dot E. So, if you see E, then you will be going to an item E dash producing E dot. So, E dash producing E dot. So, that is one possibility and from these item, if you get an e you will come to a configuration E producing E dot plus T ok. So, that is the thing. No other rule has got a dot before E. So, they will not be coming in the set I 1. So, in; so, now, in the set I 1, you see after dot there is no non terminal. So, naturally I told nothing will be added even if I take closure of I 1.

So, I one remains this set only. What about I 2? So, I 2 is on T. So, on T if we; so on T if we go so similarly we will get E. So, this should not be E dash. This should be E. This should not be E dash, it should E. So, E producing T dot and from this rule, I will get T producing T dot star F and again the same thing that we do not have any non terminal after dot. So, the closure does not have anything; only these two item survive and then I can say on so any other symbol like this open parenthesis is there. So, F producing dot E; so on open parenthesis, so it will come to a configuration where a producing open parenthesis dot E closed parenthesis.

So, this is the item and since now you have got a dot before this E. So, you have to again scan the grammar rules to see what may be the situations. So, they as you know that the grammar had this production rule E producing E plus T. So, this has got dot E. So, I have to add this E producing dot E plus T in the closure set; then E producing dot T in this set and as soon as e producing dot is there. So, T producing dot T star F will be there, T producing dot F will be there, F producing dot within bracket E will be there and F producing dot id will be there. So, in this way you can construct the set I 4 ok.

So, this way for all this from a particular state. So, you can define your Goto's and on different terminal and non terminal and accordingly, you can derive new states that the parser can go.