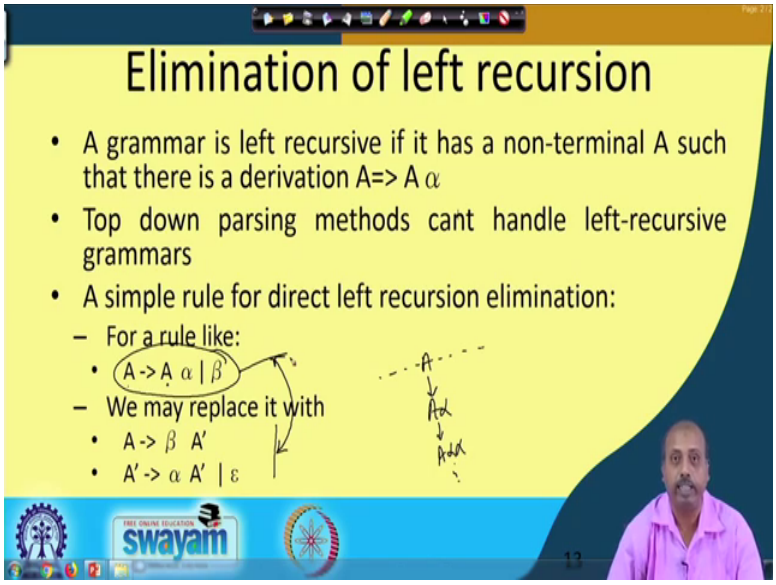
Compiler Design
Prof. Santanu Chattopadhyay
Department of E and EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 19
Parser (Contd.)

In our last class, we have been discussing on left recursion. And left recursion is a problem particularly for top down parsers because top down parsers the way they work is that whenever it that takes a decision, how to expand in non-terminal; it looks on the right hand side of the non terminal. So, whichever terminal symbols comes at the beginning, so based on that among the alternatives of a non terminal so, it will select one option.

(Refer Slide Time: 00:43)

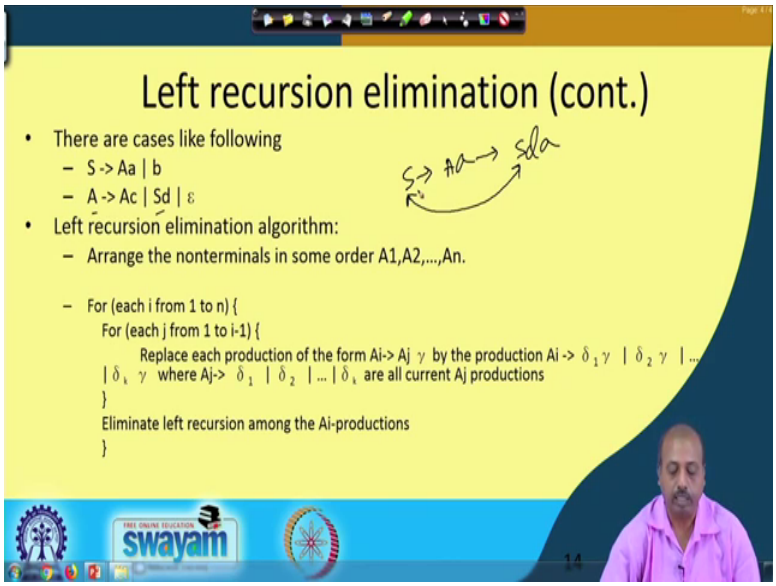


For example; if we look into this particular production, A producing A alpha or A producing A alpha or beta where alpha, so here what happens is that this A again appearing on the right hand side for the first production. So, it cannot take a decision whether to proceed with a producing A alpha whether to if in a string I have got say somewhere in the string A appears. So, in the next level, so whether this should be replaced by A alpha or not. So, that is not known because this will go on in a recursive fashion because the after this A alpha it will be replaced by A alpha, alpha and it will go on like this without consuming any further input.

So, it will be expanding like that. So, that creates the difficulty. So, what is done is that we have to resolve this left recursion and in our last class, we have seen that we can replace this particular production by means of a pair of production; A producing beta A dash where A dash is a new non terminal that is introduced into the said v n and A dash produces alpha A dash or epsilon.

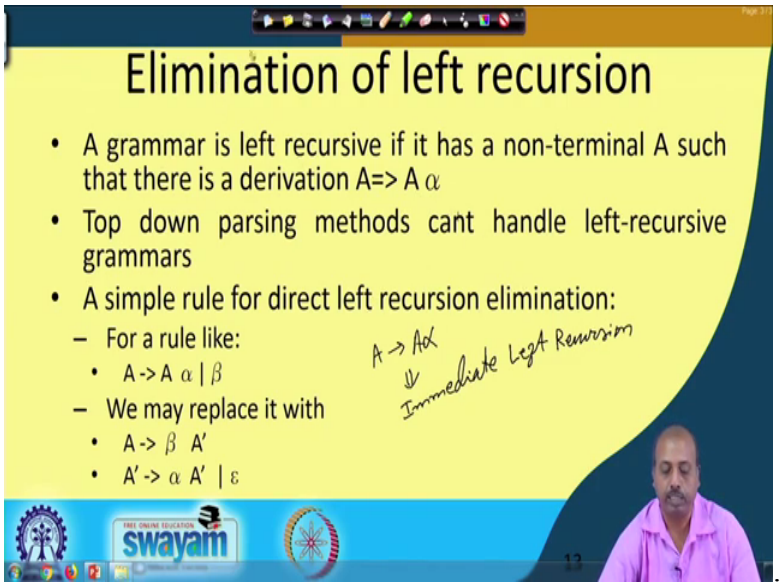
So, we have seen that these two are equivalent. So, this single rule and these pair these pair of rule involving A and A dash they are equivalent. So, today we will first look into an algorithm by which it will be; by which will be able to solve resolve this left recursion from a grammar.

(Refer Slide Time: 02:11)



So, we will be able to remove this left recursion completely. So actually problem the type of left recursion that we have seen in the previous slides. So, here this is known as immediate left recursion.

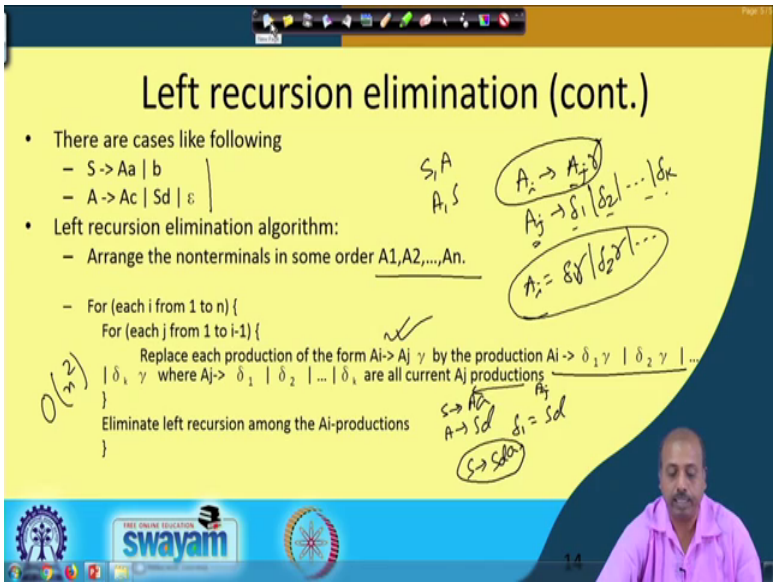
(Refer Slide Time: 02:33)



So, this A producing A alpha this the A producing A alpha so, this is known as immediate left recursion because it is appearing immediately. So, in the immediate right hand side of A, so this is appearing, but it may be the case that it is it does not happen immediately, but when it goes via a number of a non terminals or number of rules, it jointly they create this left recursion. So, we will see one example for that. So, here you see that S producing Aa or B and A producing Ac or Sd or epsilon. So, here this, so, this left recursion this A, S producing Aa. So, there is no immediate left recursion here, but after that. So, I can do an expansion like this that S produces Aa and after that I can apply this rule a producing Sd, so, giving Sda.

So, effectively the left recursion has got re introduced into the set of rules. So, not only immediate left recursion, so immediate left recursion can be resolved using the strategy that we have seen in the previous slide, but this type of; this type of collective or non immediate left recursions. So, they also need to be eliminated. So, for that purpose, there is an algorithm that we can frame. So, this is for; this is for left recursion elimination.

(Refer Slide Time: 04:17)



So, what it does is that it first arrange the non terminals in some sequence A 1, A 2, A n. So, in this particular case, we have got two non terminals S and A. So, you can order them arbitrarily. So, there is no preference of ordering so, somebody may order like this, somebody may order like this but it does not matter whatever way we ordered it.

Then it says for i equal to 1 to n, for j equal to 1 to i minus 1 so, first i equal to 1. So, it will be picking up the first production rule and if there is a there is a production like a producing A j gamma. So, will be whenever we have got a producing A j gamma will be replacing it by the production A j producing delta one gamma delta 2 gamma etcetera where A j produces delta 1, delta 2, delta k.

So, if you have a production rule like A i producing A j gamma and after that you have got a set of rules like A j producing delta 1 or delta 2 or up to delta k. If this is the situation, then wherever you have got this A i producing A j gamma. So, this rule has to be replaced by these set of rules. So, A i producing delta 1 gamma; delta 1 gamma or delta 2 gamma etcetera up to delta k gamma.

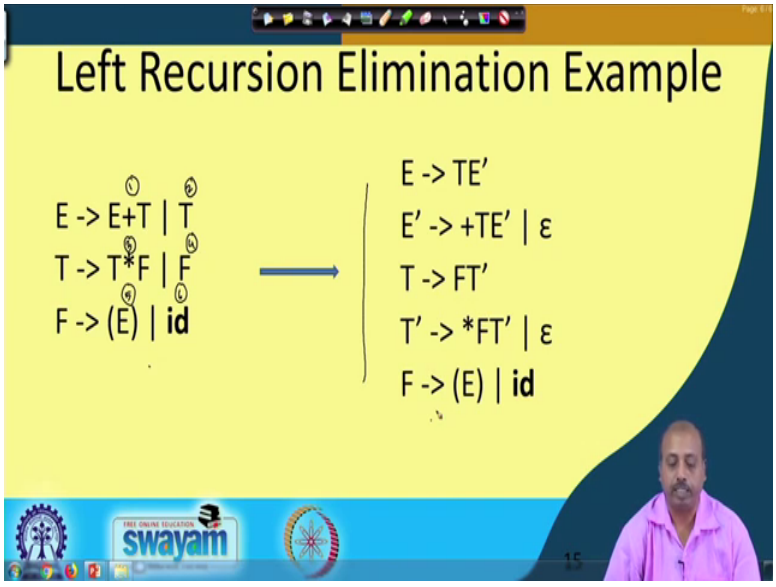
So, after that, so essentially what we are doing we are replacing this A j here by this delta 1, delta 2 etcetera. So, this process may introduce some immediate left recursion. So, after doing, this process may immediate introduce immediate left recursion like here we have got say S producing Aa and there is a production rule which says A producing S d

where this AaAa. So, this is our Aj so, this with respect to this particular rule, so, this capital A is our Aj and our delta we have got only delta 1, delta 1 is S d.

So, if I follow this rule, so what it will what will happen is that. So, S producing A j will be replace by S d and gamma is A ok. So, as for as this is example is concerned gamma is A. So, it will be replace like this. So, it will bring in immediate left recursion. So, if there is any in the set of rules then it will eliminate this left recursion by the technique that we have seen previously as we have seen in the last slide.

So, that way of for all the; for all the production rules, so it will eliminate the left recursion. So, that, the resulting grammar it will not have any left recursion as for as complexity of this algorithm is concerned. So, this is you can understand that the this is be go n square time algorithm because where n is the number of the production rules that we have in the system. So, let us take an example and try to see how this left recursion elimination can work.

(Refer Slide Time: 07:29)

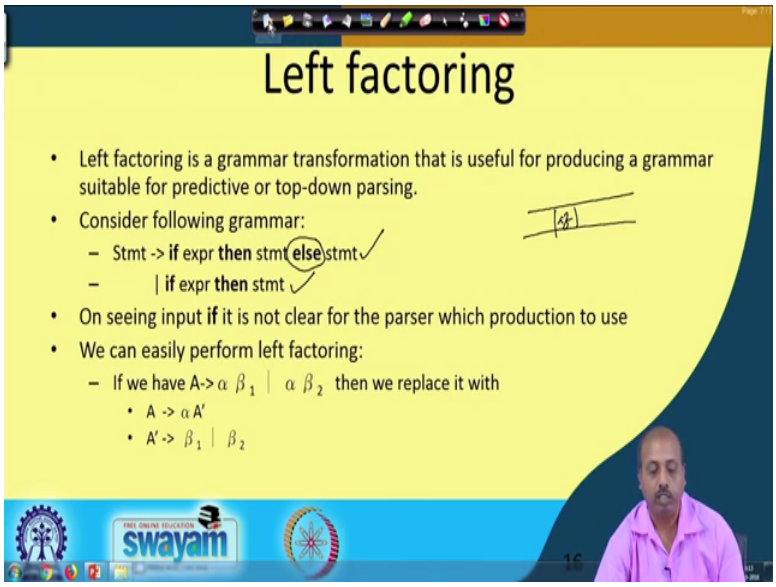


So, this is the standard ETF grammar; E producing E plus T or T, T producing T star F or F and F producing within bracket E id. So, if we number the rules, so this is my E producing E plus T. So, this is rule number 1, E producing T is rule number 2, T producing T star F is rule number 3, T producing F is rule number 4 and we have got this as rule number 5 and this as rule number 6.

Now, you see we have got here E producing E plus T. So, so here, so this will be this is having a immediate left recursion. So, this will be replaced by this set of rules E producing T E dash and then E dash producing plus T E dash or epsilon. So, this is by immediate left recursion. Similarly we have got T producing T star F. So, T star F again this is having and immediate left recursion so, that will that way it will be release. So, in this particular grammar, so we do not have any non immediate left recursion. So, all the left recursions that we have they are all immediate in nature.

So, we do not have any; we have we do not have any such production rule which will introduced some new left recursion by the by means of substitution. So, here everything is immediate so, and this right hand side, so here all the rules we have eliminated the left recursions from the immediate case and as we have seen previously also like this grammar and this grammar they can generate parser tree for the same expressions for the they actually represents the same language. And as for as the left recursion is concerned, so we have seen that they are also doing the same thing the grammar will the power of the grammar does not change or the set of language is accepted by the grammar does not change by doing this left recursion elimination ok.

(Refer Slide Time: 09:39)



So, next we will see another important issue which is known as left factoring.

So, left factoring is another problem like if we have got a situation like say this or we have got a rule where statement produces; if expression, then statement; else statement or

if expression, then statement now when you have. So, in your input sequence suppose I have got the token if.

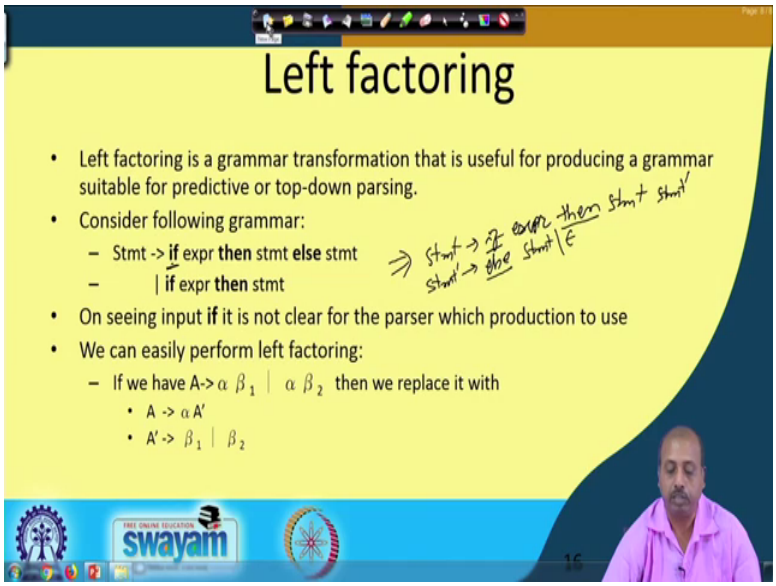
The lexical analyzer has return to the parser that the next token is if. Now, the parser has to take a decision whether it should follow this rule or this rule. So, if it expands via this rule; that means, it is assumed that in future else we will definitely be there, but if this trans out to be a normal if then statement it does not have any else part. So, later on the parser will be in a in problem. So, it will not be able to parse that particular statement.

On the other hand, if it assume the parser assumes that the statement will be if expression then statement, then we have the difficulty that if actual statement is if then else statement, then later on when that else token will be written by the lexical analyzer to the parser. So, parser will not be able to accommodate that. So, in any way, so since the we are going in a top down fashion. So, all this left factoring left recursion this problems occur for the top down parsers because, top down parser have to has to take a decision looking on the next token that it has receipt which rule to follow.

So, getting the token if the parser cannot decide whether it should follow the if then else rule or the if then rule. So, this is another important problem and this has to be resolved. So, left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parser parsing so, this is an example. So, on seeing input if it is not clear for the parser which production to use. So, what is the way out? Way out is that we take up to this much. So, up to statement part so we take it as a proper rules and this else statement part will take as another rule so that the parser up to this much a single rule will follow and after this if else comes. So, a new rule will be used it is not that I have to take a decision looking at the first if itself.

So, or so, this is the example like if we have this alpha beta 1 or alpha beta 2. So, far as per as this example is concerned this alpha is if expression, then statement beta 1 is else statement and beta 2 is epsilon. So, what we do, we replace it by a producing alpha a dash ok. So, up to whichever is common so, take it take it out. So, alpha A dash and A dash producing produces beta 1 or beta 2. So, that is the obvious thing that we can do. So, in this particular case, so we can write it like this.

(Refer Slide Time: 12:41)

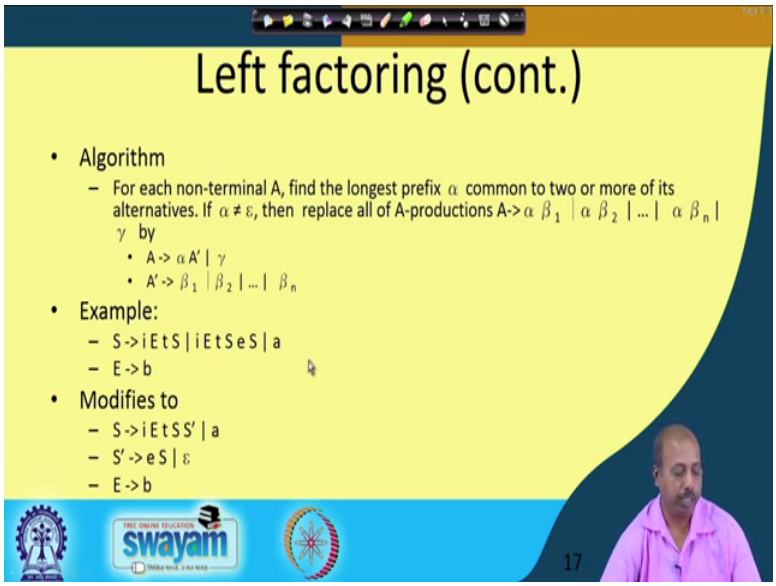


Statement produces if expression; if expression, then statement and then statement dash where statement dash produces else statement or epsilon ok so, do it like that. So, one thing one notation that will follow throughout this lectures is that whenever we are writing in say in type font like say whenever writing in the slides. So, wherever we have got a terminal symbol we have tried to put it in boldface like this to distinguish it from the non terminal symbols.

Like here expression is a non terminal, but if is a terminal similarly statement is a non terminal. So, we will try to write this if then else this tokens in bold face and other terms like expression statement etcetera which are non terminal symbols, they will try to write in normal font. So, whenever the it is whenever there is a scope of confusion. So, if there is no scope of confusion, then of course, any font can be used. Similarly, when writing in by our hand. So, you will underline the tokens like this if then else. So, these are tokens so, we will underline the tokens by to distinguish them from non terminals we will underline the terminals to distinguish them from non terminals wherever it is confusing say if it is not confusing, if it is straight forward, then we will follow the normal convection ok.

So, the rule followed is you take out this alpha part separately the alpha A dash and then A dash produces beta 1 or beta 2. So, you can do it like this so, that is left factoring.

(Refer Slide Time: 14:43)



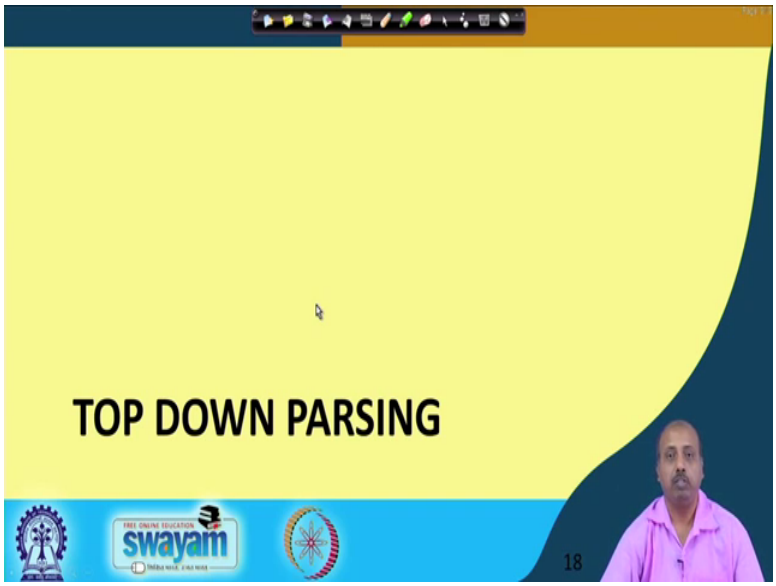
So, this is the general algorithm for left factoring for each non terminal A, find the longest prefix alpha common to 2 or more of it is alternatives. If alpha not equal to epsilon, then replace all A productions by A producing alpha beta 1, alpha beta 2 up to alpha beta n by this one; A producing alpha A dash or gamma and A dash produces beta 1 or beta 2 or beta n.

So, this simple rules so, takeout the longest prefix between a number of; between a number of alternatives and take them out. So, this is that if then else gamma that is written in short. So, S produces if expression then statement or if expression, then statement else statement or a and E produces b.

So, where a b these are terminals indicating some expression and all. So, it that is not a concerned here, so here this there is a left factoring of this parts i E T S. So, this part is common between this first rule and the second rule. So, this grammar can be modified to something like this S produces if expression then statement dash S dash or a where S dash produces e S that is else S or epsilon and E producing b. So, this part remains as it is.

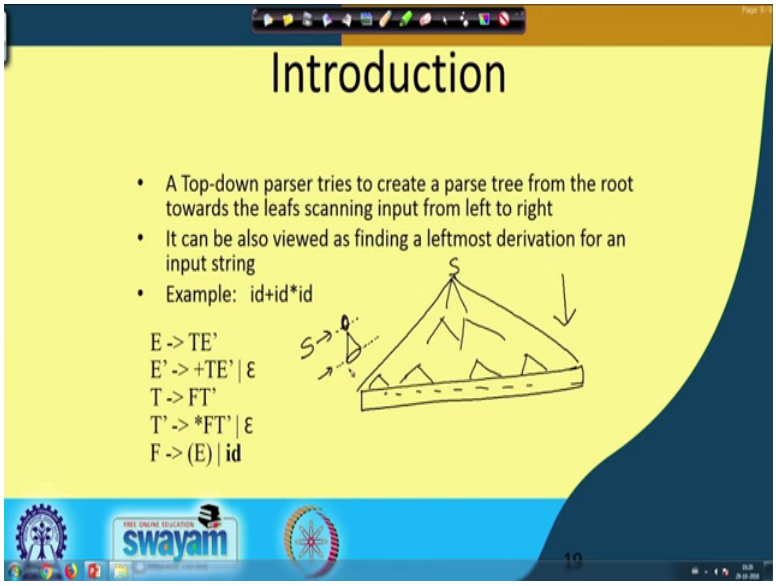
So, this is the way we will do this left factoring. So, that the grammar becomes suitable for top down parsing or predictive parsing.

(Refer Slide Time: 16:23)



So, next we will be looking into one of the parsing strategies known as top down parsing. So, top down parsing it will try to derive the whole stream whole given stream starting with the start symbol of the grammar and in the process it will generate the parse stream and that parse stream may be used for other purposes, but it will be it will try to come from the top it is starting with the start symbol of the grammar, it will try to derive the whole stream.

(Refer Slide Time: 16:55)

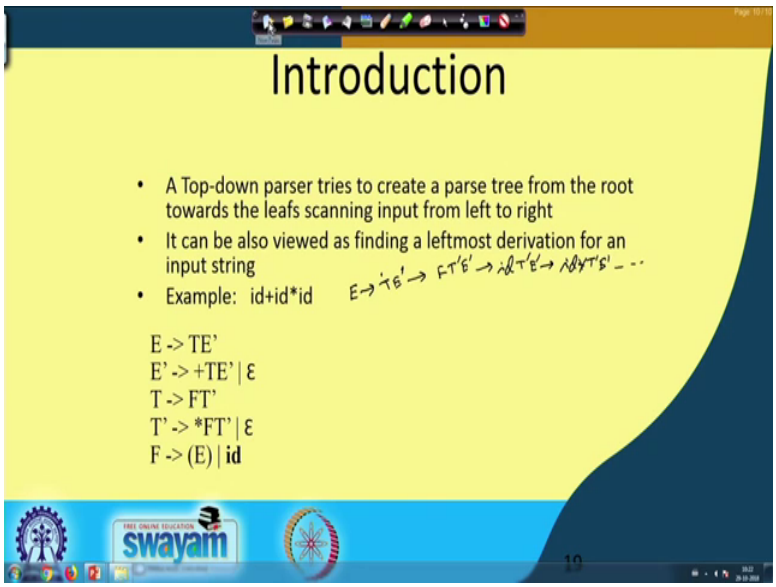


So, how it will do? So, we will see that. So, a top down parser it tries to create a parse tree from the root towards the leaves scanning in foot from left to right. So, if this is the so, you can conceptually view it like this if this is your input stream; if this is the input stream that you have. So, it will be ultimately covered by tree whose root will be the start symbol and it will be going like this.

This entire tree will get covered and ultimately, at the lowest level you will have this thing and this is done in a top down fashion ok. So, that is a top down parser. So, it can also be viewed as finding left most derivation for an input string. So, this perform say left most derivation so, it replaces the left most non terminal symbol by means of a terminal symbol. So, it is by means of a corresponding rules so, it starts with S, the starts symbol then uses some rule to produce some go towards the input string. Then wherever is the first non terminal symbols, suppose this is the first non terminal symbol, so in the next line next state so, this thing will get expanded. So, this part will remains. So, this will get expanded to something like this and this will be there then again it will find out the left most non terminal and it will expand it.

So, it will it does a left most derivation of the input. So, suppose I have got this id plus id star id. So, in a leftmost derivation how will it be done?

(Refer Slide Time: 18:37)

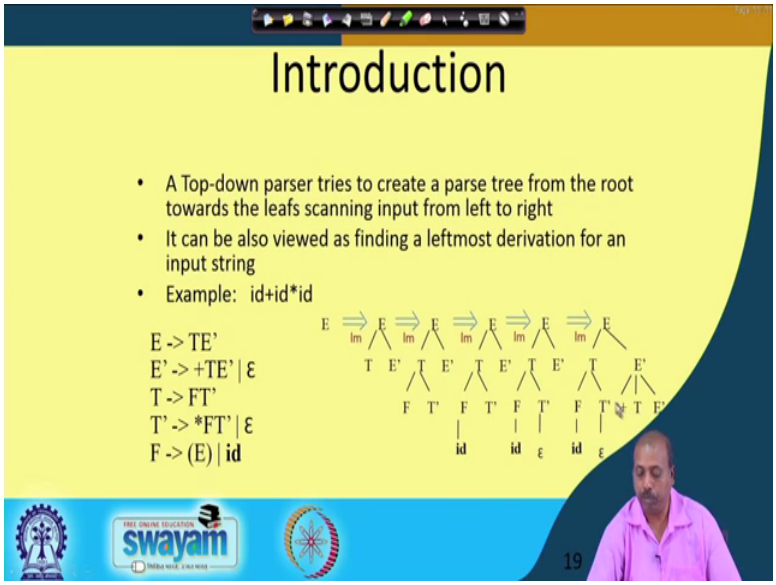


So, it will be starting with E as the start symbol, then there is only one rule that involves E. So, it is T E dash ok. So, T E dash, so we have got, so next terminals in the next

production; so I have to replace this T because this is the leftmost non terminal. So, this T will be replace by F T dash. So, it because F T dash E dash and in the next rule this F will be replaced and F will be replaced by id so, it is id T dash E dash.

Then this next this T dash has to be replaced then this T dash has to be replaced by this rule T dash producing star F T dash. So, this is id star T dash E dash so, it will go like this and ultimately, it will derive this thing. So, that is. So, we will see some examples again but it will go it will proceed like this and do the whole derivation.

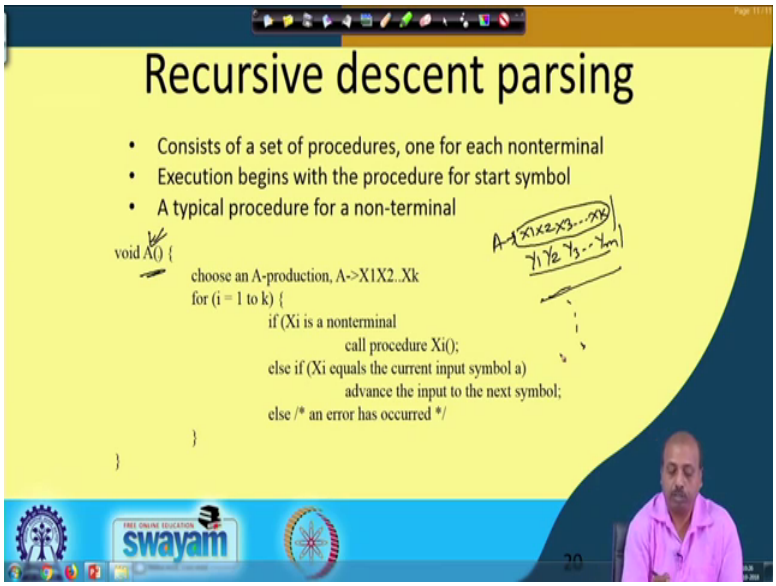
(Refer Slide Time: 20:07)



So, this shows the derivation E producing in a left most derivation, T E dash then in the next step this T is replaced by F T dash and E dash in the next step, this F is replaced by id and this is T dash, then the in the next step this T dash is; T dash is producing epsilon. So, that is it is replaced by epsilon and then this T produces so, this part is done now this it goes to this E in this E it this E dash is expanded by plus T E dash and then it will go on like that. So, after that it is not shown here.

So, this plus T up to we have derived up to this id and this is plus T E dash. So, it will be going on like that producing the whole string. The rest of the derivations are not shown here ok.

(Refer Slide Time: 21:05)



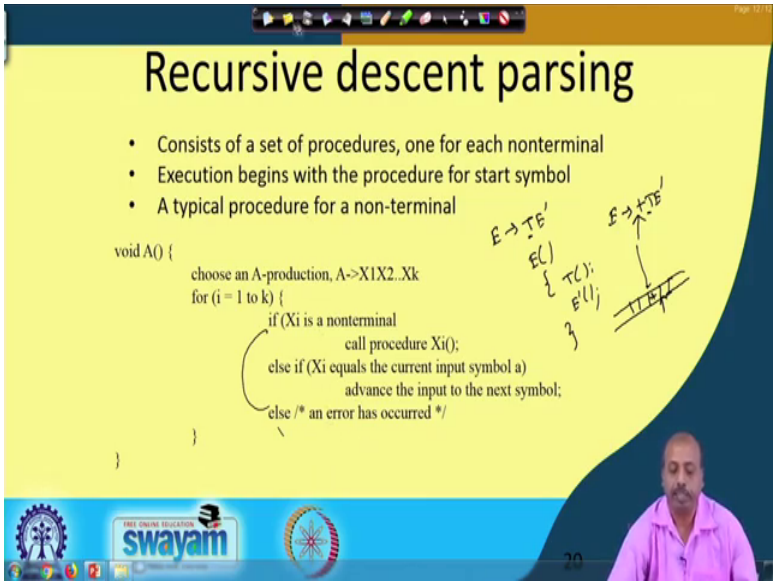
So, there are two top down parsers that will look into. The first one in the category is known as recursive descent parsing. So, in a recursive descent parsing it consists of a setup procedure. So, as the name suggests, so it is a recursive parser. So, it is a collection of a recursive routine and it will try to recursively descent. As I said, that it starts with a start symbol of the grammar and tries to derive the remaining strings the final string. So, it actually in some sense it descents from the start symbol towards the final string. So, that is why it is called a descent parsing and it is recursive descent parsing because it does a recursively by means of procedures which are recursive in nature.

So, it consists of a set of procedures one for each non terminal. So, for every non terminal, we have got a procedure. Execution begins with the procedure for start symbol so, whatever be the start symbol so, with that there execution will begin. So, from the main routine you call give a call to the procedure corresponding to start symbol of the grammar, a typical procedure will look like this.

Suppose, A is a non terminal symbol in the grammar, so, A is a non terminal symbol in the grammar. So, for that we have a corresponding procedure A and no parameter needs to be passed here so, because it is and it does not return any value also. Now, this a may have several alternatives like my production rule may be A producing x 1, x 2, x 3, x k or y 1, y 2, y 3, y, y m. So, like that there may be several alternatives.

So, what this parser is trying to do it is try, it will first try to see whether using this rule it can reach to the final string or not. If it fails, then it will try with the alternatives next alternative. If it fails, then it will try with the next alternatives. So, that way if all the alternatives are exhausted and the final string cannot be derived then there is an error in the input. So, input the syntactically wrong input is syntactically wrong otherwise it will say that it has got a derivation for the input string starting with the start symbol of the grammar. So, it will do like this. So, it will for a for a particular non terminal A it will select A production rule A producing X 1, X 2 up to X k and for i equal to 1 to k, if X i is a non-terminal, it will call the corresponding procedure X i.

(Refer Slide Time: 23:57)



So, as I said that say this particular rule, so, E producing say T E dash for example, so this is a rule. So, what it what will happen is the in the procedure E, it will first call the procedure for T, then it will call the procedure for E dash so, this is the whole routine. So, that is what is written here for if X i is a non-terminal like T is a non-terminal here, then call the procedure X i. So, it will call the procedure T.

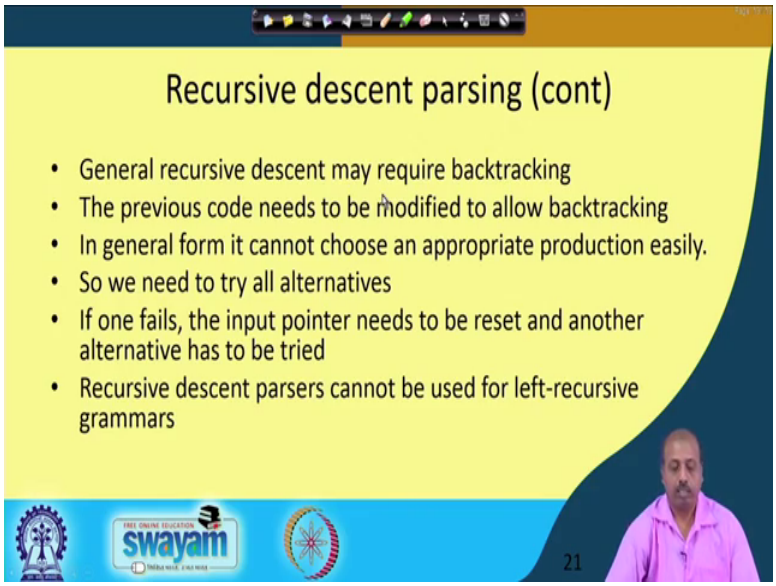
When it returns from the procedure T, then it will come to this point. Now, I advances to the next point so, now, it is E dash so, it will call the procedure E dash. So, that is for the non terminals. So, if X i happens to be a terminal; for example, if I have got a rule like E producing say plus T E dash in that case, it will first this X 1 happens to be a non-terminal terminal symbol plus, then it will check with the current input ok.

The lexical analysis tool, so the lexical analyzer will be informed by the parser to get the next token. So, if the lexical analyzer also finds a plus at the input point, then it will return the token plus and the parser will see that these two are matching ok. So, if it is a terminal, else if X i is a terminal symbol the input symbol a, then advance input to the next symbol. So, if X i equals the current input symbol a so, this is also plus this is also plus, then we advance the input pointer to the next point.

So, now, so naturally now, it will be call now it comes to this loop again and as a result it now it will be calling the routine for T. So, it will go like that otherwise there is an error. So, there is an error that has occurred so, it will be flashing that error that there is a syntax error.

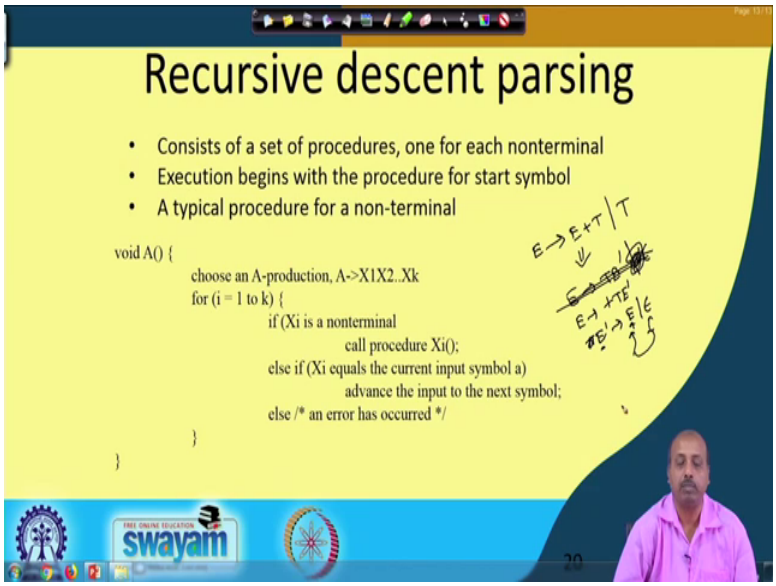
So, we are not showing it for explicitly for all the rules. So, this will go for all the rules. So, this is a generic way we have written it. So, it will be there for each and every non terminal that you have in the grammar. So, you see that this recursive this and parsing algorithm is very simple. So, you have to just have a collection of recursive routines and then those recursive routines. So, they will be they will be collaborating with each other to see whether the input string can be derived from the start symbol of the grammar.

(Refer Slide Time: 26:39)



So, general recursive descent parsing may required backtracking. So, this is one important issue as I was telling that I may have several alternatives like say A producing X 1, X 2 say E of.

(Refer Slide Time: 26:57)



For example, I can have two rules like E producing E plus T or T or say if I if solve the immediate left recursion, so this will be giving rise to E producing T E dash or epsilon. So, E producing T E dash and sorry E producing plus T E dash where E dash produces E or epsilon say something like this. Then I may have some alternative like if may say E dash for the rule for E dash. So, it may call the routine for E or it may be calling it may do nothing.

So, that way I can have several alternatives. So, if this alternative fails, then I will be go walking with the next alternatives. So, that way I may have to have different recursive calls and one so, one if one of the alternatives fail, then I have to call recursively the next alternative. So, the this is, this is that is why it is recursive in nature but this recursiveness create some problem also because, implementing recursive programs sometimes it is difficult. Now, it takes more time, then a non recursive version. So, we will see how to go to a non recursive version but first to understand the concept. So, this recursive parsing, so that is very easy and given a grammar, so you can very easily make it.