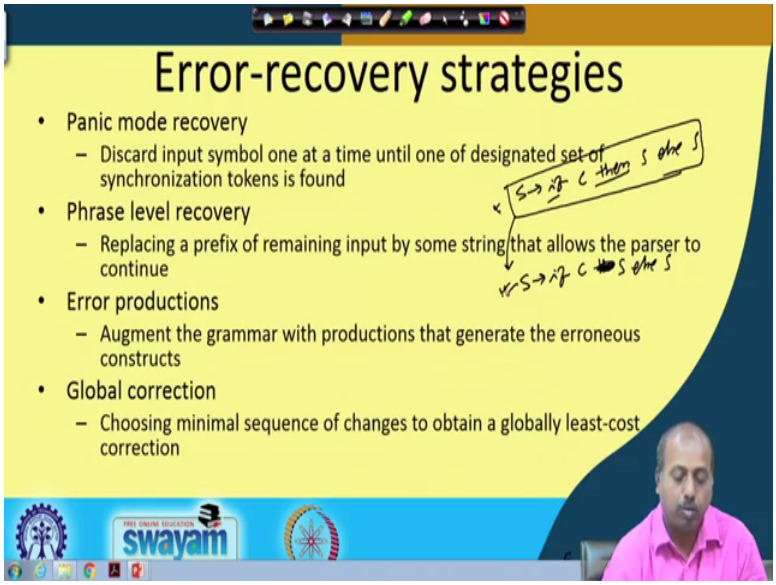
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 18
Parser (Contd.)

Another type of error recovery strategy is by means of error production. So, error production says that the language designer has given as a state of production rules that tells how the valid statements of the language can be designed, can be generated. Now, compiler designer can think about the possible errors that the user can do and accordingly add some few more productions into the system which will be called error productions.

So, if a particular, going back to that example that we are taking of if then else statement so it is.

(Refer Slide Time: 00:53)

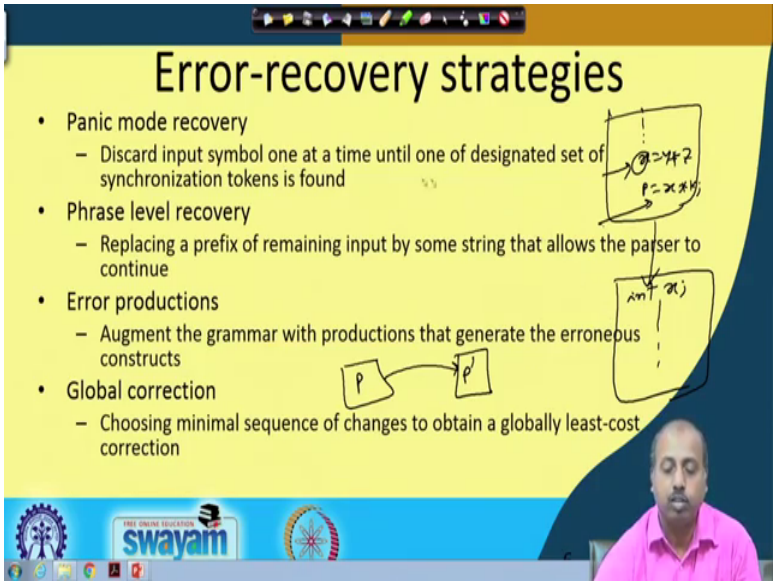


So, it was like this the S producing if some condition then statement else statement. Now, this is a rule from the language, this is a production rule from the language. Now compiler design can interpret or can try guess that the type of error, type of mistakes that the user can do is forgetting these then.

So, accordingly the compiler designer can add another rule to this language. So, if C S else S then what will happen if a program has got a line where we have got this where the then party is missing then what will happen is that this rule will not match, but this rule will match, but the compiler designer knows that this is a error production. So, it should not try to generate code based on this.

But the parser it will be able to proceed with this rule and it will not fall into an error condition. So, that is a basically used in the error recovery of the parser. So, this is the error productions. So, you can augment the grammar with productions or that generate the erroneous constructs. So, this is the other possibility and also you can do some global correction. So, we can find out some minimal sequence of changes to obtain a globally least cost correction. So, this is a very I should say costly approach. So, what it is trying to do is that there is an input program.

(Refer Slide Time: 02:38)



So, there is an input program say P and there is that has got some errors in it. So, what it will try to do is to transform this P into if another program P dash where all those errors have been corrected and that connection is global. So, for example, if you have got say in your program suppose you are using a variable x and at various places in the program so you are writing like x equal to y plus z, P equal to x into k like that and if this variable x is undefined then you see that at every place it will generate a message that x is undefined, x is undefined like that. So, you can transform this program into another

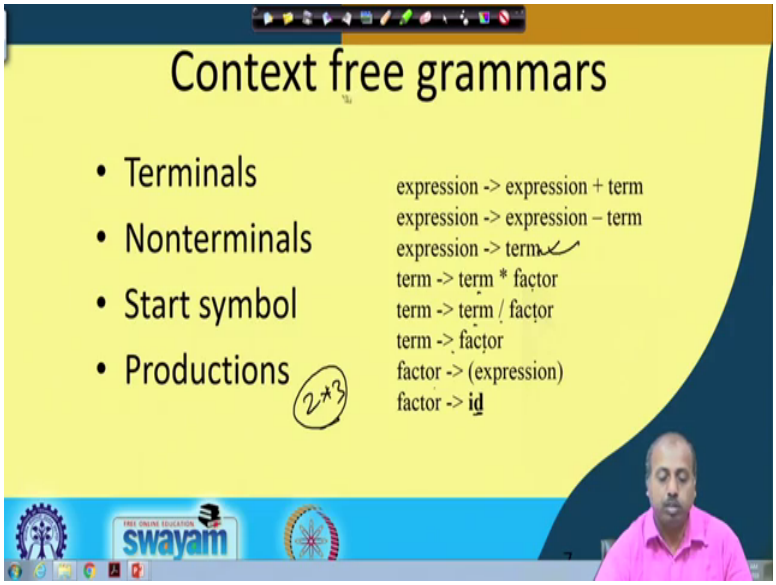
program where at the beginning you guess what is the type of x, since I am doing this addition and multiplication integer operation. So, I can define x as integer ok. So, this line may be introduced into the program.

Rest of the thing remains as it is, only this extra line has been introduced. So, what will happen this program will not generate any semantic error. So, this that way the parser will be able to process the compilation process will be able to proceed or say. So, x will get installed into the symbol table as a result it will that lexical analyzer will be able to return proper token that x is an identifier token so like that. So, it will help in the compilation process, but there can be many ways there can be many interpretations.

So, why do not we the replace x by a constant number. So, that is also a correction. But so, that way what is the minimal number of changes that can be done in the program to obtain a globally least cost correction. So, that is the challenge of this global correction strategy. So, this is particularly useful where you have got text formatting type of application where, we try to show the output to be a as correct as possible to the user and if the user is not satisfied with the output we will perhaps go to the correction.

But many times what happens is that the correction done by the formatting tool is good enough and we do not plus we do not do any further corrections. So, this global correction has got applications in text formatting type of cases.

(Refer Slide Time: 05:04)



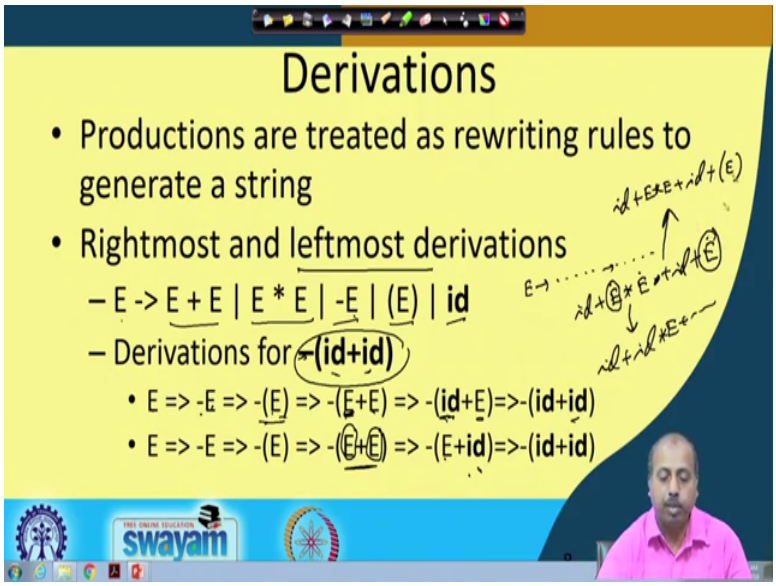
So, next we will be looking to the in more detail the grammar. So, this is coming back to the context free grammars. So, as I was telling that most of the programming languages so they have got this context free grammars, they can be specified in using context free grammar. And, as I said that it has got four set series terminals, non-terminals, a special start symbol which is a which is a member of the set of non-terminals and the set of production rules.

So, this is a typical example of a grammar. So, this is known as expression grammar. So, throughout our course many times we will be referring to this particular grammar. So, we read it like this, expression produces expression plus term then or expression produces expression minus term or expression produces term. So, term is basically product of two sub expressions and expression can be sum of two sub expressions or the sum or subtraction of two some sub expressions or it may not have any such addition subtraction.

So, if it is say and only products. So, 2 into 3 in that case this whole expression is considered as a term using the third rule. And, then this term is taken as it is take a term is can be term multiplied by factor or term divided by factor or a term can be simply a factor and factor can be within bracket expression or a or a single identifier. So, this way we have since previously some derivations.

So, this is a grammar and we will be calling this as expression grammar and it takes care of expressions, arithmetic expressions having the operators like plus minus multiplication division and over the parenthesis. So, it takes a it can detect parenthesized expressions involving identifiers and the arithmetic operators plus, minus, multiplication, star and slash addition, subtraction, multiplication and division. So, this is a good example and very often we will be referring to this example.

(Refer Slide Time: 07:24)



So, derivation; so we have already seen something called something like derivation like see suppose we have got an expression like this a grammar like this. So, it has got a set of rules if producing E plus E or E star E or minus E or within bracket are id fine. So, this is the grammar. So, we can have a derivation for this one, say suppose this is the expression given to us minus id plus id. So, how can we derive it? So, we can do it like this first we can we apply this rule.

E producing minus E so that this unary minus is taken care of then this E is replaced by, this E is replaced by within bracket E. So, this is open parenthesis this is close parenthesis this is parts have been generated, then this E is replaced by E plus E is because I need to generate this id plus id, then this first E is replaced by id and in the next case the second E is replaced by id. So, this is one possible derivation of the string minus of id plus id starting with the start symbol of the grammar E.

Now, somebody may say that I have a choice at this point. So, when I am looking into this E plus E I have a choice whether I can replace this E by id or I can replace this E by id, in the first case we have replace the first E by id. So, there can be another derivation where the second E is replaced by id first and in the next generation the first E is replaced by id.

So, in general I can say that if I have got a string ok. So, I have started with the start symbol of the grammar and from there I am driving the strings like this, than at any point

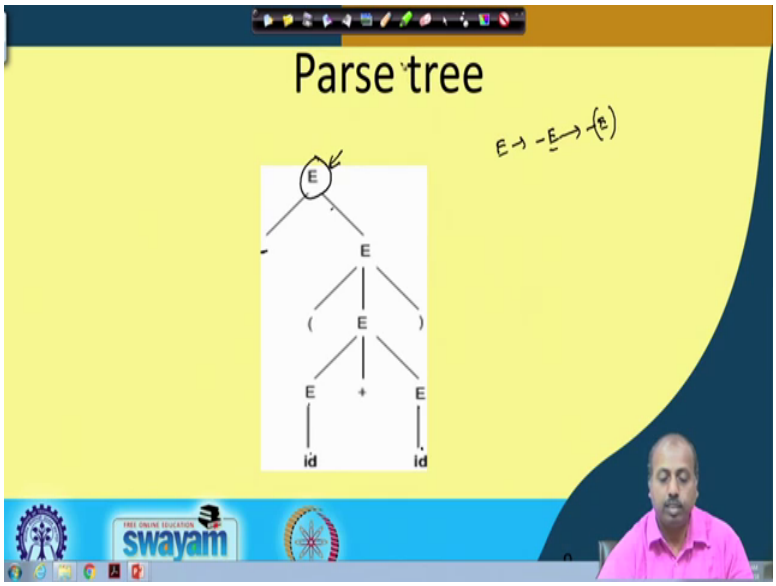
of time suppose I have got some string like id plus E star E plus id like this plus E, from this I want to generate my next string ok. Now there are possibilities like I can replace this E I can replace this E or I can replace this E I have got three choices.

Now, there are two derivations that have been that that is followed into a by the compiler designers, one is that you replace the leftmost non terminals that is this one. So, that is known as leftmost derivation. So, in the next derivation so if I write it like id plus this E producing id star E etcetera etcetera. So, that will be a left most derivation because, I have replace the left most non terminal by the right hand side of the corresponding production rule or I can replace this E.

So, I can have the derivation like E star E plus id plus this E producing say within bracket E that can be done. So, I have replaced the last E the rightmost E in this case. So, we can have two different derivations, one is called a left most derivation another is called a rightmost derivation. So, in the rightmost derivation from the intermediary string the rightmost non terminal symbol is replaced by the corresponding right hand side of the production rule and in the leftmost derivation the left most non-terminal is replaced by the right hand side of the corresponding production rule.

So, these are this is known as derivation. So, productions are treated as rewriting rules to generate a string. So, they are they are basically productions are nothing, but rewriting rules. So, we can replace one string by another string by applying those rules and there are derivations like right most derivation and left most derivation.

(Refer Slide Time: 11:37)

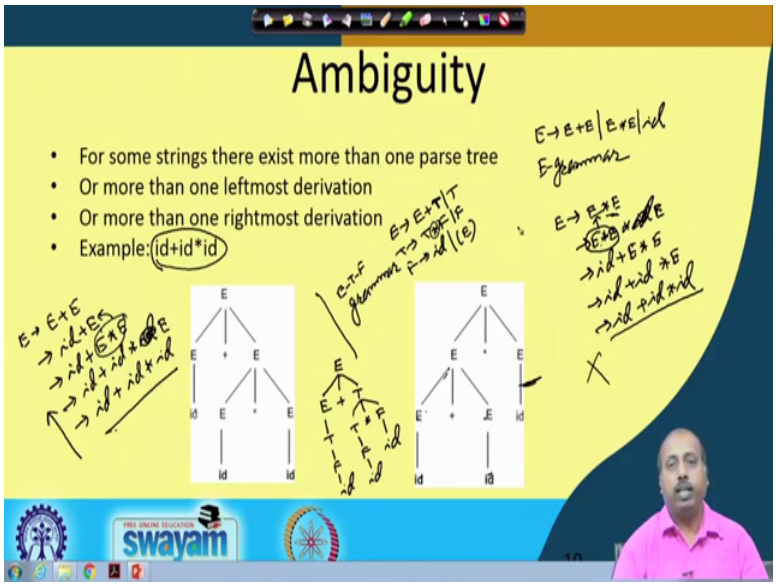


We can think about parse tree like the derivation that I have drawn I have done in the last slide. So, you can think of it to be like this. So, E is the start symbol of the grammar and from there we have got; E is the start symbol of the grammar and then I have replaced the rule E E I have use the rule E producing minus E.

So, minus so it has got two children nodes, one is the minus the terminal symbol minus another is the terminal symbol E and the next step this E is replaced by within bracket E. So, it so the derivation that we had is minus within bracket E; so, this E is replaced by within bracket E. So, that is the next part of the tree, now from this I can have this E producing E plus E. So, this E is replaced by E plus E. So, that is and the next phase of this tree that we have, next level of the 3 E plus E and the next level this E produces id and this E produces id. So, this way I can have a tree representation of the derivation.

Where the first derivation that I have done, so it constitutes the first part of this tree and the route of this tree is the start symbol of the grammar and from there the rules that I am applying. So, based on that we can have different we can have different branches in the tree. So, this is the parse tree, parse tree type structure.

(Refer Slide Time: 13:21)



Next we will have sometimes so there can be ambiguities in a grammar for example, suppose we consider this particular specific this particular statement or example id plus id star id. Now, I am following the grammar, I have a I have got a simple grammar like E producing E plus E or E star E or id since I do not have parenthesis and all in this particular string. So, I do not need to consider the other constructs that the grammar may have, other rules that the grammar may have. So, we have got only these rules. So, one derivation for this id plus id star id may be like this, we start with E then replace it with E plus E then the first E is replaced by id second is replaced by E star E and then this E is replaced by id and this E is replaced by id.

So, this is one particular tree, another parse tree can be like this. So, here I first I replace this thing by E E star E. So, instead of doing it like this so in plus. So, I am starting with this star. So, E producing E star E and then this E it produce, it gives me E plus E and this E gives me id this E gives me id and we have got. So, the derivation sequence in this case is E giving E plus E from their giving id plus E from their id plus E star E from their id plus id star id. So, id plus id star E and this is giving id star id.

And in this case what has happened? I have started with E and from there I have got E star E, from there I have got this E replaced by E plus E. And then star id star E. So, let us do it like that it is then this E is replaced by id plus E star E then this is giving me id plus id star E and that will give us id plus id star id. So, both in both the cases I have got

the finial string id plus id star ids in both the cases and both of them are doing left most derivations ok. So, here also I am substituting the left most non terminal first in the sub string. So, here also I am doing the same thing. So, if I do that, but even. So, both of them are parse trees corresponding to left most derivations, but even if I do that you see that we have got two distinct parse trees for the same strings of the language. Now which one is correct?

So, sometimes so, whenever you have got more than one such derivations. So, we say that the grammar is ambiguous in nature because it has got more than one interpretation of the same string. In fact, if you will be you if you look into it more carefully then in this case you see that here this if you if you go in a bottom of fashion then this is the id so this is the id is replaced by E and then E star E is replaced by e. So, you can understand that this expression multiplications is done first and then it is added to the first id.

Whereas in this case this id plus id. So, they will be the so this thing is the. So, ultimately this E plus E is done first. So, addition will be done first and then the result will be multiplied by this is E plus E is giving E. So, as if this addition is done first and then it is multiplied by the second E. So which is, if we considered the presidents of this addition and multiplication; so, multiplication has got higher presidents then addition. So, naturally this is not a valid interpretation or of the statement, but as long as the presidencies are not mentioned. So, both of them are correct it ok.

So, the language designers I have they have they have to tell the presidents and as if compiler designer either you have to design the grammar such that this ambiguities are not there ok. Will, we can see shortly like if I have got that E T F grammar where I have got the production rules like E producing E plus T or T then T producing T star F or F and F producing id or within bracket E.

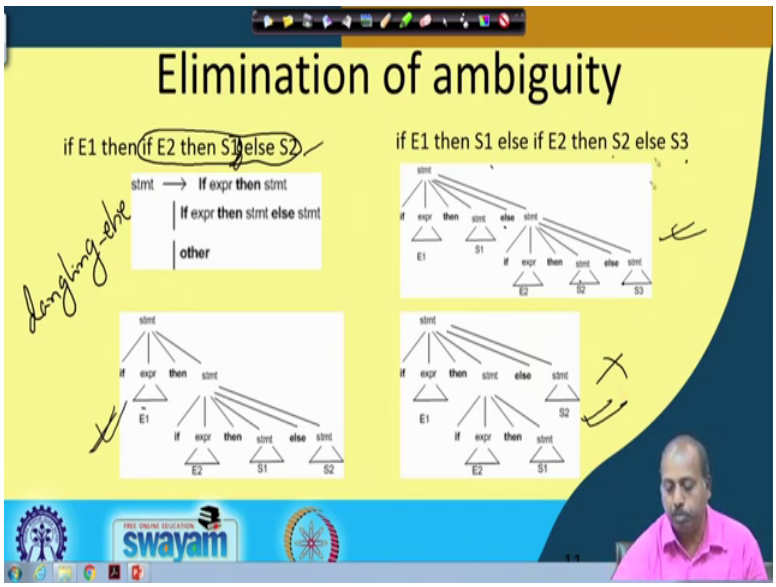
So, this particular grammar so, this is known as E T F grammar E T F grammar. So, this E T F grammar does not have this ambiguity E T F grammar does not have any ambiguity because it says that for reducing for applying this rule first this t has to be derived. So, this E plus so and the T we will take care of this multiplication first. So, addition cannot be done earlier than the multiplication. So, this will be more clear when we go to the code generation chapter, but this we can just take if view that this E T F

grammar cannot be cannot be ambiguous and. In fact, you will not be able to derive this parse tree using E T F grammar.

For example if we use E T F grammar then you will always you have you always have do it like this E plus T then this E giving T giving F giving id and then this T giving T star F and this giving F and this giving id and this F giving id. So, this is the only parse tree that you can have. So, you cannot have any other parse tree for this E T F grammar, on the leftmost derivation. So, it does not have any ambiguity whereas, the this particular grammar where this T and F non terminals are not there so they are known as E grammar or expression grammar and this is ambiguous.

But many times this ambiguity is kept because in this case I have three non terminal symbols needed for writing the grammar et and f and here there is a single non terminal and you will see that the size of the parser. So, it will vary it will be more if there are more number of non terminals. So, though ambiguity is a difficult, but it is a problem for this parsing process. So, many a times you will allow this ambiguity so that the size of the parser is less compared to the situation where you have got a totally unambiguous grammar. So, you will see as we as we proceed through this chapter in the successive portions of this course.

(Refer Slide Time: 21:22)



So, this is another source of ambiguity like say this if then else statement. So, suppose we have got this statement if this rule like it says that a statement is like this, that if E 1

then if E 2 then S 1 else S 2 ok. So, the corresponding grammar is this one. So, it says that a statement can produce if expression then statement or if expression then statement else statement or other statement. So, this is where this is other actually it encompasses all other statements that may be there. So, we are not bothered about other statements. So, we have just pet kept it as other. Now suppose we have got suppose we have got this thing that if E 1 then if E 2 then S 1 else S 2 now. So, this one can be derived like this. So, this if E 1 so this is the expression.

So, it follows the first rule if expression then statement, then the expression will be giving me E 1 and this statement is again broken down into if expression then statement else statement where this second expression part. So, this corresponds to this E 2 and this S 1 and S 2 so they are corresponding to then part and else part. So, another possible derivation of the same thing can be like this, say the I have by start with statement now if expression. So, I take the second rule. So, for the first level of derivation I take the second rule and it derives like statement producing if expression then statement else statement and for this expression gives me E 1 that is fine. Now this statement gives me this part.

If E 2 then S 1. So, this part so it does it produce a like if expression then statement and E 2 and S 1 fine. So, so this is the situation. So, what has happened is that in this case, in the in the first case this in the first case this else S 2 part this else S 2 is the confusion ok. So, in the first case in this diagram so this has been taken with this particular if statement as you see if E 2 then S 1 else S 2 whereas, in this case this S 2 the else part has been taken with the outer if ok. So, it is it has been taken like this that as if I have got this whole thing as a as the then part ok.

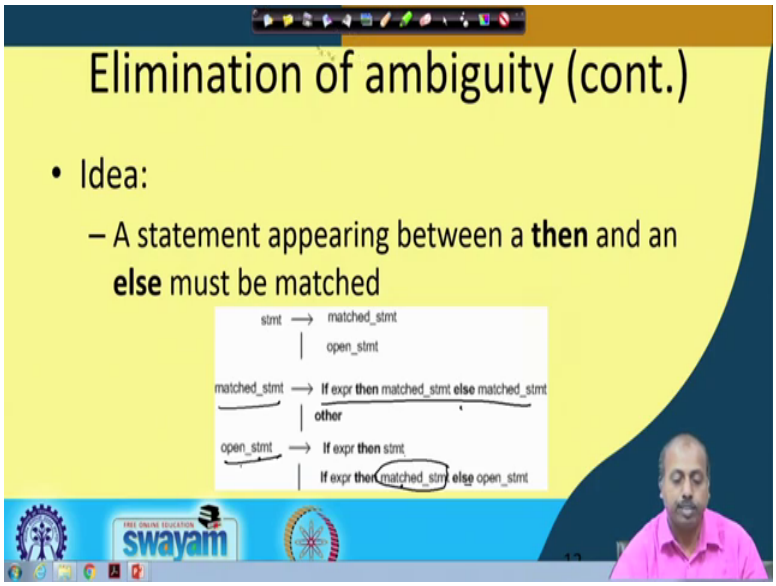
If E 1 then this statement else S 2. So, if E one is false it will go to S 2 whereas, in this case if E 1 is false it will go to the next statement and if E 1 is true then it will come to check E 2 and then it accordingly it will go to S 1 or S 2. So, there is a the two different derivations are possible for the same statement, whenever you have got this type of nested else we have got this particular problem. What about this case like if E 1 then S 1 else if S 2 then S 2 if E 2 then S 2 else S 3. So, here I have got in the then part I have got E 1 and S 1 else and in the second part I have got if E 2 when S 2 else S 3.

So, here I do not have any confusion because there was no. So, the here the problem occurred because one of the, if statement had has the else part the other one does not have any else part. So, in the nested if where does this else go so that is the ambiguity, that is the problem. So, in one case in the first case we have taken else with the inner motive in the second case here. So, we have taken else with the outer motive if, but if there are too such else's. So, both the statements are if condition then statement else statement and nested like that then there is of course, no problem. So, this is always fine. So, it there is no confusion here, but this is the confusion and you remember that most of the programming languages they will tell that this else is always attached to the innermost if.

So, this is known as the problem of dangling else this is known as the problem of dangling else. So, so the most of the programming languages why which allows this type of nested if then else statement it will tell that the if it is a mention if it is written simply like this then this else S 2 is always part of the innermost if so it is actually like this the. So, the statement to be taken as if E 1 then this whole thing; so, this is this is valid, but this is not valid ok, but as far as the grammar is concerned both of them are valid ok.

So, we have to do something to the grammar. So, that it is taken care of. So, you do not have the problem of this type of dangling else. So, otherwise the parser we will find a confusion here and there are several ways by which it will report it and as if proceed through the portion you will see that there are different ways in which this will be reported by the parser I am sorry.

(Refer Slide Time: 27:08)



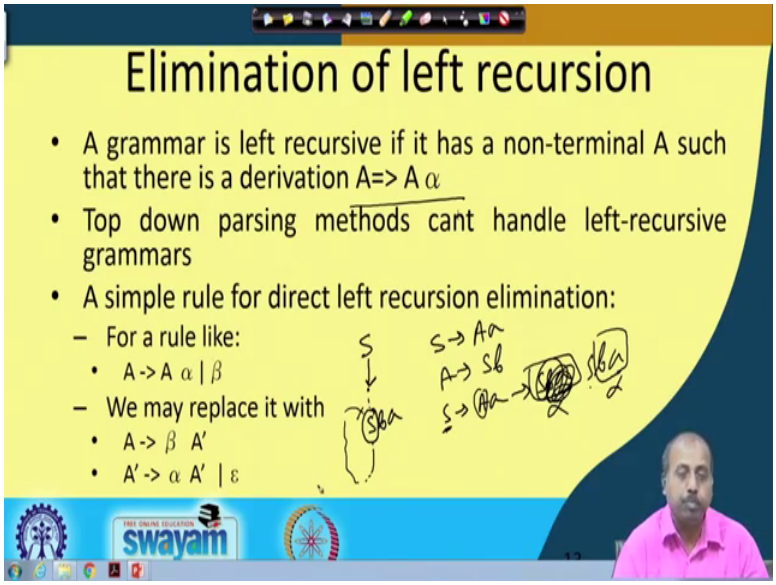
So, next will be looking into how to eliminate this dangling else problem; so, it one possibility is that a statement appearing between a then and then else must be a matched statement. So, this is the thing that thus I we say that there are two types of statements, one is called a matched statement another is called an unmatched statement or open statement. So and man a matched statement so it can give me all other statements of the programming language, but as for when it comes to the if part sorry, when it comes to the if part then it is always if then else.

So, I do not have the if expression then statement as the producible from the matched statement. So, so if expression then statement can be produced only by the open statement. So, with the open statement can also produce if expression then statement else statement, but this first then it must be a matched statement. And so that this else becomes a part of this. If the else is there then it will become a part of this if it is not there so, it will be a matched statement so that if there is an else inside. So, it will be matching with the innermost if. So, that way this solves this dangling else problem.

Now, so if we if we do some derivation then there was. So, this in the previous diagram what will happen is that. So, this part; so this part can be produced only by matched statement. So, as a result I cannot have this type of structure. So, only this structure will be possible because this will be a matched statement this is open statement and this is the

matched statement and the matched statement we will have the else part with it as a result it will not be able to derive the first string.

(Refer Slide Time: 29:23)



So, next will be looking into some work that talks about eliminating the left recursion. So, many a time a grammar has got rules such that starting with a non terminal you can derive a string which is A A. So, which is which is having again that-non terminal as the left most symbol. For example, if I have a grammar rule like this say S producing say A a and there is a rule A which says that it can give me S b, then what will happen if you apply this rules then this gives me A a and from this a is again replaced by S.

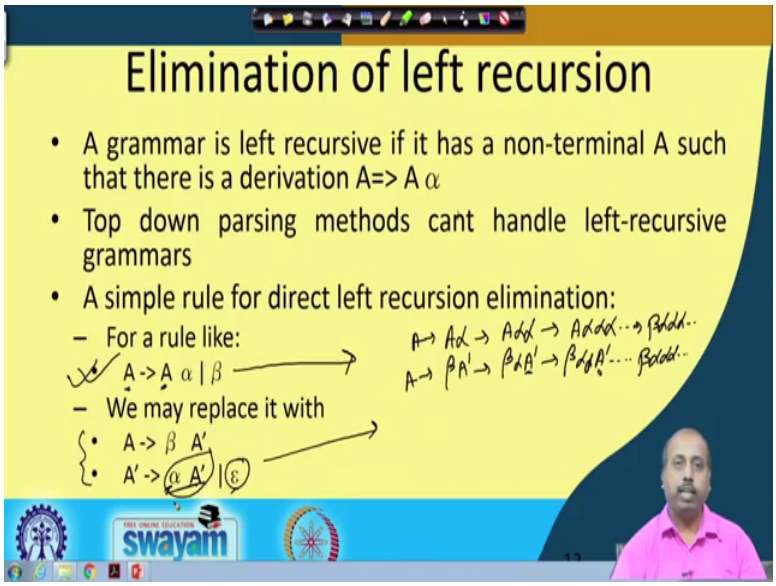
So, S b a, so what is happening; so, starting with A you have produce so, this whole thing is an is alpha. So, alpha is a string of terminals and non-terminals. So, the first character sorry this is not alpha. So, alpha is this part. So, this is giving me S ba. So, this ba part is alpha. So, as per this convention A producing A alpha. So, this S is giving me S alpha. So, that way the symbol the non-terminal symbol that is there on the left hand side is appearing again as the first symbol in the right hand side. So, as if in some sense it is you and you can understand this is the some type of recursion ok

That is the same non-terminal symbol appearing as the first symbol on the right hand. So, this is the parsing methods that we have so, parsing methods cannot handle this type of left recursive grammars because, per the top down top down parsing methods what they will do? It will start with S and it will try to derive a string from here say in after

sometime it comes to the situation where it is S ba. So, basically we it is back to this is S. So now, again so, it will fall into a loop here and it will try to do that again. It will try to again apply the same set of rules to replace S and that way it falls into an infinite loop.

So, if you are looking for parse the top down parsing strategies then this left recursion is a problem. So, there can be some modification, simple modification to the grammar that can convert a left recursive grammar to a non-left recursive grammar. So, the rule is like this suppose, we have got a rule like this A producing A alpha or beta where alpha beta are strings of terminals and the non-terminals. So, we can replace it by a producing beta A dash and A dash producing alpha A dash or epsilon.

(Refer Slide Time: 32:21)



So, you see that whatever strings that can be derived from the first grammar that is this rule can also be derived from this rule like here. So, you can see that A can produce so, all the strings that this A can produce.

So, A can be applying it can go on producing A alpha. So, it can do it like this A alpha alpha A alpha alpha alpha. So, like that so, it can do it like this and ultimately this alpha has to be replaced by beta. So, it will finally, give a string which is beta then alpha alpha whatever. So, you see that using the second rule also so, you can do like this. So, you start with A first we do beta than A dash and this A dash can give me beta alpha A dash and then again this A dash can give me beta alpha alpha A dash. So, I just go on applying

this rule and finally, this A dash can be replaced by this epsilon. So, that you have got the beta alpha alpha alpha like that ok.

So, whatever string is derivable from this rule is also derivable from this set of rules, but the advantage that we have is the second set of rules it does not have any left recursion in it ok. Unlike the first rule where there was a left recursion, the same non terminal was appearing as the first symbol on the right hand side. So, it does not happen in this case. So, here it is simply that situation does not occur. So, the top down parser it can take a decision whether it should follow rule number 1 or rule number 2 that is whether to follow this rule or this rule for A dash.

So, it will see that if the remaining part of the string it starts with alpha then it will follow this rule, if it finds at the it is the end of the string then it will apply the second rule a dash producing epsilon to resolve the issue. So, that way the top down parser will be will not will not get stuck at that point, it will be able to progress. So, this way this a left recursion elimination is a very fundamental step whenever we are going to discuss about top down parsing strategy. So, this is going to be one of the most important thing that we should do, you should scan through the grammar and find out where this is the left recursions are there and you eliminate all those left recursions from the grammar.