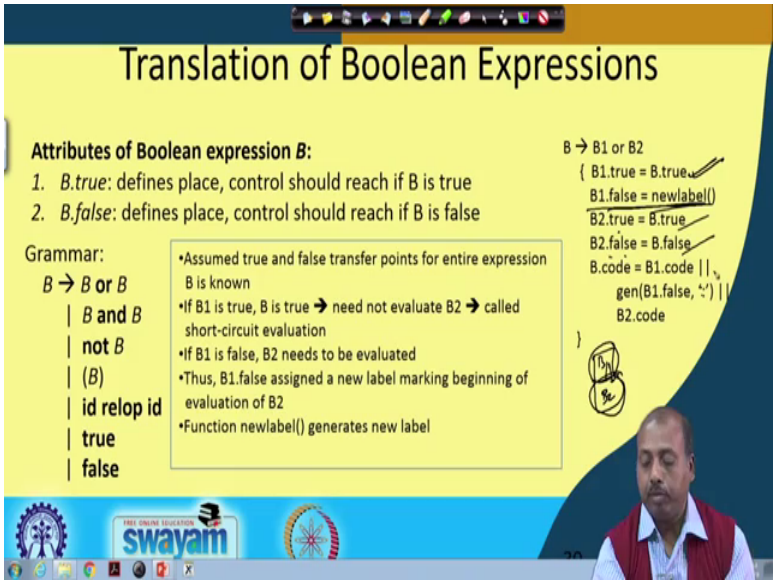
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture - 55
Intermediate Code Generation (Contd.)

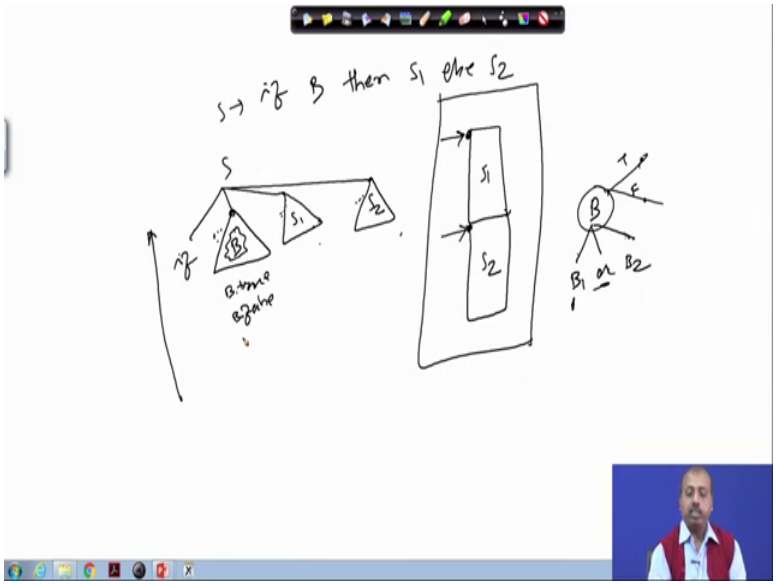
So, for example, if you look into the first rule of this of this grammar B or B.

(Refer Slide Time: 00:19)



So, for the sake of our understanding so, we are writing it as B, so B producing B 1 or B 2. So, like this now we will assume that the true and false transfer points for the entire expression B is known. So, it is part of so, B is part of some bigger expression like say. So, let us do it like this.

(Refer Slide Time: 00:49)



Say if some Boolean expression then I have got a block of statements S 1 else S 2 where S 1 is a block of statements and S 2 is another block of statements. Now if you look into the parse tree for this so, it will be something like this S producing. If then a portion of the parse tree which will correspond to B then there will be a portion of the parse tree which corresponds to S 1 and there will be a portion of the parse tree where I will have this S 2, fine.

Now, it is assumed that when I am doing this reduction. So, if B is true it will so, so since all these reductions have already been done. So, this S 1 and S 2 position in the three address code file. So, they are known is not by that time the codes have already been generated. So, I can very easily tell what is the offset of the code S 1, what is the offset of the code S 2 in the full three address code file, the file that contains the three address code. What are the offsets of these positions? So, naturally I can assume that for this B this B dot true and B dot false. So, these two pointers where they should point to. So, that is known when I am going to do this reduction.

Similarly, if I, so, just to extra polite that fact. So, if I have got a rule like this. So, B 1 or B 2 so, we know that when I am doing this. So, for this B so, we already know the true and false pointers for that.

So, where the true and false should go, but then for now if now what the situation is that if B 1 happens to be true then since it is an or operation. So, if this is the true branch and

this is the false branch then B 1 being true, it should follow that true branch of B and B 1, so B 2 B so, similarly B 2 being true. So, it should follow the true branch of B and similarly if B 1 is false and B 2 is false, if both are false then only it should go to the false branch.

Now, it is a bit confusing because I said that when I am doing this reduction. So, in this particular case when I was doing this reduction, this targets were known where S 1 and S 2 are in this file that are known, but here my B expression is not yet over. So, I do not know where the where so, I am in the, I am actually parsing somewhere here bring the parsing at this point. So, at that time I have not reduced the S 1, I have not reduced S 2. So, as we are looking into the numbering policy of this reduction. So, you see that these numbers that are appearing for this. So, they are much less compared to this numbers and this numbers. So, they have not yet been reduced. So, they are that they will come later. So, naturally for a single parse type of operations. So, it is difficult, but if we assume that my compiler the code generation phase will be in a two pass fashion.

So, in the first pass, so you can identify all these jump targets, ok. For B dot true B dot false. So, that the second pass while generating the actual code so, you can just use those say, you can just use those addresses. Off course, you can do something else also that we will see for single pass compilation. So, we need to do some sort of back patching type of operation, so that we will see later. So, for the time being, it is assumed that we have got this whenever we are having this particular rule B producing B 1 or B 2. So, it is assumed that the true and false transfer points for the entire expression B are known. Now if B 1 is true B is also true. So, we need not evaluate B 2. So, B 1 appears to be happens to be true then B 2 did not be evaluated without evaluating B 2. Also, we can say that the whole expression B is going to be true.

So, this is called a short circuit evaluation, because we are not evaluate B 2, if B 1 already true. Similarly, if B 1 is false then off course we cannot tell anything, because now I need to evaluate B 2 also. And if B 2 is also false then it will be then we have to follow the false label.

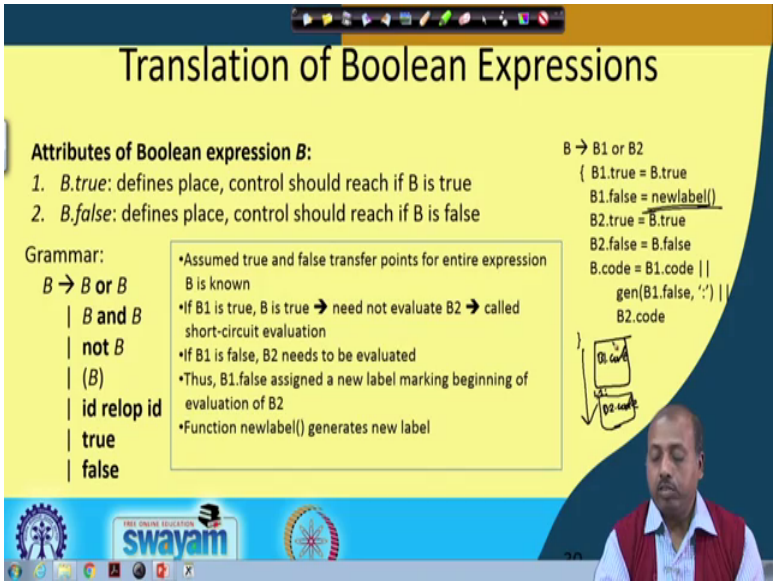
So, the way the code is that the semantic actions the, the syntax directed actions are written is like this that beyond the true is made equal to B dot true. So, since, you assume that we already know B dot true. So, B 1 dot true will be like that, but for B 1 dot false, I

do not know where to go, ok. So, that way I generate a new label. So, this new label is say L 1 so, B 1 dot false equal to L 1. Now B 2 dot true equal to B dot true B 2 dot false equal to b dot false, because I will first have the code for B 1, then I will have the code for B 2.

So, if B 1 is not true then it has run like this. So, it has run like this and then I will be so, this B 2. So, at the end B 2 dot false will be equal to B dot false and this B dot code, it will have the code for B 1, it will have the code for B 2. And then, but in between there will be a label that will get inserted.

So, here you see that the B 1 dot false. So, you have got the label. So, that will be inserted here. So, B 1 dot false colon. So, that will be inserted then I will have got the code for B 2 dot I will have the portion for B 2 dot code. So, what I was the situation final situation for B dot code will be something like this.

(Refer Slide Time: 06:57)

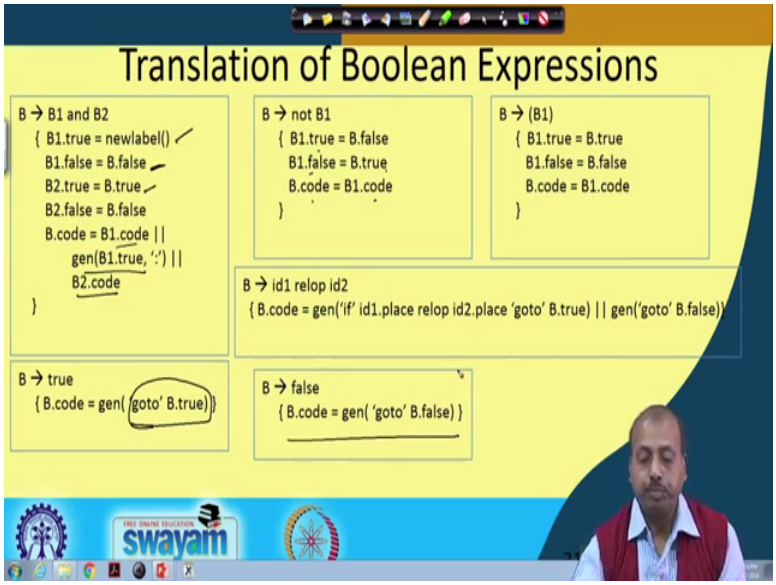


So, here I will have got the code for I will have the portion for B 1 dot code, this is the B 1 dot code and after that I will have a, this newlabel that is there. So, this newlabel L 1 has been generated. So, this is so, B 1 dot false contains the label.

So, this is L 1 suppose the newlabel that is generated is L 1 and after that it will be the B 2 dot code will come. So, after B 1 dot code, this L 1 will be generated this L 1 label will be generated then this B 2 dot code will be generated.

So, it will be going like this. So, this after this B 1 dot code is over see this label will get inserted and then this B 2 dot code will be put at the end. So, so this way I can generate the code for this B 1 or B 2. So, we can we look into the other rules other grammar rules for generating the code forother Boolean Expression or other rules like B equal to B 1 and B 2.

(Refer Slide Time: 08:11)



So, in that case if B 1 is true then I have to evaluate B 2 also, but if B 1 is false then it is a short circuit. So, if B 1 is false. So, B 1 dot false is made equal to B dot false then B 2 dot true is. So, B 1 dot true is newlabel then this B 1 dot false is B dot false so; that means, it is a short circuit. And then B 2 dot true is B dot true B 2 dot false is B dot false then B dot code is the concatenation of B 1 dot code then this particular label B 1 dot true colon and then B 2 dot code.

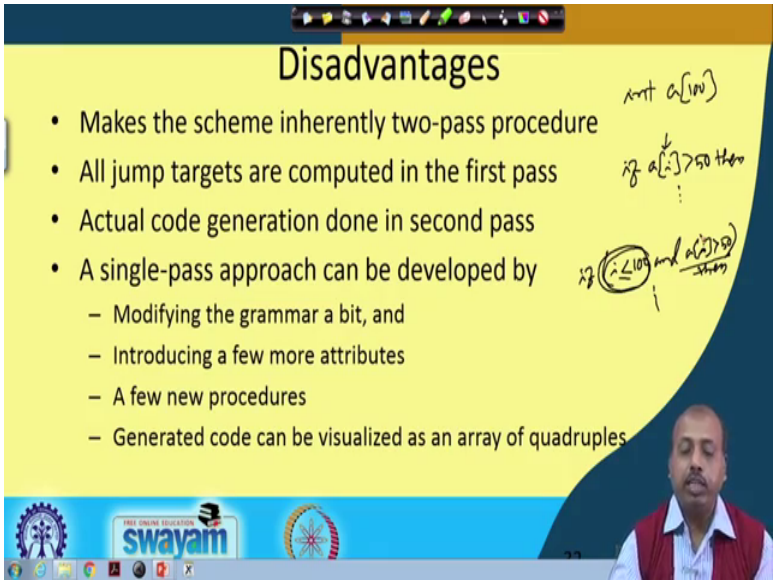
So, this that way I can have this B 1 and B 2 then not of B 1. So, B 1 dot true is B dot false B 1 dot false is B dot true and B dot code is equal to B 1 dot code. So, no new code has to be generated, but this true and false pointers.

So, they have to be reversed. So, wherever the B 1. So, B is B dot false wherever it was going. So, they, they should be we should go there if B 1 is becoming true. Now, and wherever it was going on B becoming true. So, on B 1 becoming false we should go there, ok. So, then B within bracket B 1. So, this is also very simple. So, B 1 dot true equal to B dot true B 1 dot false equal to B dot false B dot code equal to B 1 dot code.

So, like that now B producing true. So, this is B dot code, it will generate a goto B dot true.

So, then because we know it is definitely true. So, that way I do not have to evaluate anything. So, this is B dot, B dot code will be. So, this particular line will be generated goto B dot true. So, B dot true is already known the target is known. So, it will go there similarly, B dot false is like this. Now, id 1 relop i d 2. So, this will generate the two statements. So, if i d 1 dot place relational operator i 2 i d 2 dot place goto B dot true otherwise it will goto B dot false. So, that way this i d 1 relop i d 2 will be done.

(Refer Slide Time: 10:41)



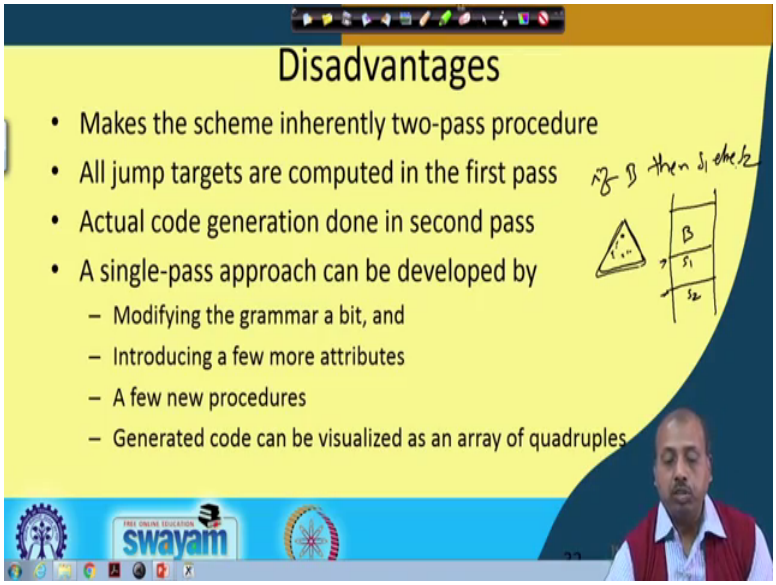
So, how does it help us in generating the generating code? So, this helps because I can have the short circuit of Boolean operators and all. So, I do not evaluate the complete expression and so, that helps, ok.

In many many programming languages you see this is helpful like see suppose I have got this shape. So, I have got an array in my programming integer a 100 and at some point of time I need to check whether if a i is greater than 50 then something something. Now one problem with this type of situation is that. So, if this index i itself has become more than 100 the value of the expression i itself is more than 100 then whenever, whenever I am trying to do this a i access. So, this can generate some violation of condition, ok. So, the some run time error may occur. So, it is a safe practice that we write like this. If i less than 100 less or equal 100 and a i greater than 50 then I have got the piece of code.

Now, here you see that if I happen to be more than 100 then this part itself will evaluate to false. So, I will not go into this evaluation and it will. So, it will not try to access some arbitrary memory location to get the value of a i. So, that way my code execution is much safer, ok. So, this is to good like for this short circuit of this Boolean expression evaluation. So, this is really helpful; however, the way that we are generating the code it has got problems, because the first thing is that it makes the scheme inherently two pass proceeding, because in the first pass all the jump targets will be computed and in the second pass the actual code generation will be done. So, that is that is required, because I need to have this B dot true B dot false. So, those pointers their destinations are calculated before we generate the code.

However, so that makes it a bit combustion also. So, there is a modification to the grammar, so that we can make it a single pass approach. So, single pass approach, it can be developed by modifying the grammar a bit and introducing a few more attributes, ok. So, few new procedures and the generated code can be visualized as an array of quadruples. So, that way we can so, that is correcting some portions some corrections can be carried out in the code that is generated. Basically when the jump targets become known then only we can tell the address like.

(Refer Slide Time: 13:43)

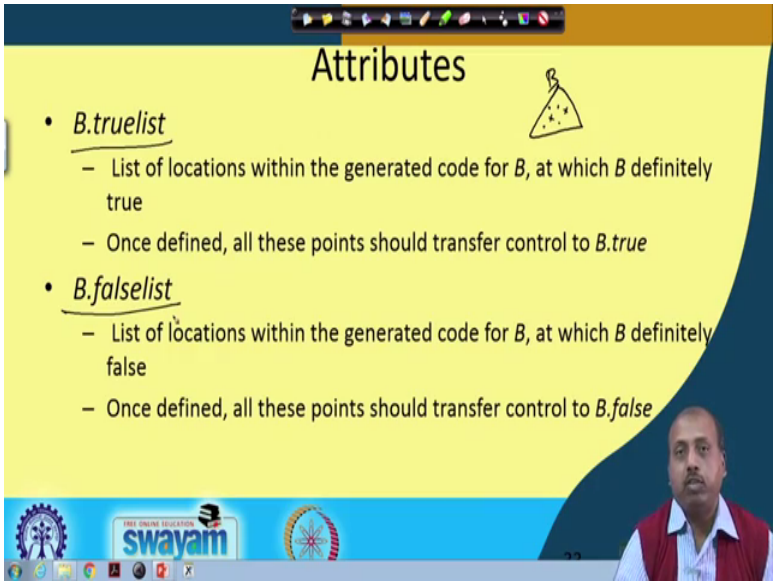


For example if I have got an statement like if some Boolean expression B then S1 else S 2. Now while parsing this B so, I have got at several points the B becomes true or B

becomes false like that. So, at those points are need to transfer the control to either S 1 or S 2 but until unless the code for B is over.

So, if this is the file where I am writing the code for this three address code. So, until unless the code for B is over I cannot start the code for S 1 and until unless the code for S 1 is over I cannot start the code for S 2. So, when I am this intermediatory points, I really do not know this particular offsets the offset of S 1 and offset of S 2, because it is all dependent on size of B 1 size of S 1, etcetera. So, I need to do something so, that in the first pass I can take it as I can know that the offset values, ok. The second pass I do that; however, in the single pass procedures. So, you have to modify something so, that we can we can generate the code.

(Refer Slide Time: 15:03)



So, how is it being done? So, the attributes that we are talking about new attributes apart from that B dot true and B dot false. So, we have got a truelist and a falselist for every Boolean expression. So, B dot truelist, it is the list of locations within the generated code for B at which B is definitely true. So, as I was telling that if this is the portion of this is the parse tree for B part. So, this is the parse tree for B. Suppose at this points in the parse tree the B expression is definitely true.

So, we put all these locations in a list called B dot truelist. Similarly there may be a few locations where B is definitely false. So, they are put on to the list B dot falselist. So, in B dot truelist, it is the list of locations within the generated code for B at which B is

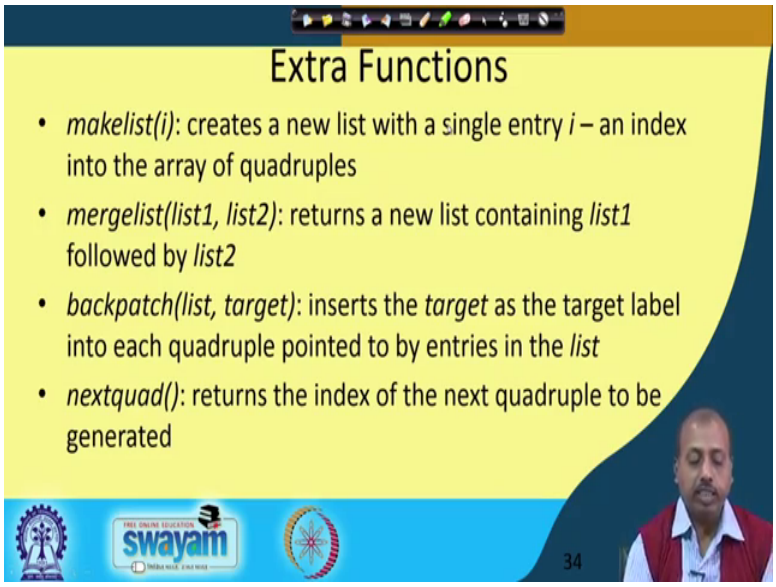
definitely true. And once defined all these points should transfer control to B dot true. So, when I am generating the parse tree for B. So, at that time I do not know what is B dot true and B dot false targets but after some time they will become defined.

So, for example, that if then else statement that, then part and else part the statements they will come after some time of parsing. So, at that point so, this jump targets will known and for this for the nonterminal B. So, we have note down the true list where all this points of all these points at which the B is true. So, that that values have been kept.

So, at all those places, we can replace the address the jump address that can that was left as blank at that point. So, they can be filled up with this jump this new address which is this B dot true or the address of S 1 in if then else statement similarly B dot falselist. So, the it contains the list of locations within the generated code for B at with B is definitely false and once defined all these points should transfer control to B dot false.

So, they should I should have the all controls transferred to B dot false. So, all these points I can do some correction in the target code that is generated. So, that they are rectified and now they contain the address of B dot false as their jump target.

(Refer Slide Time: 17:25)



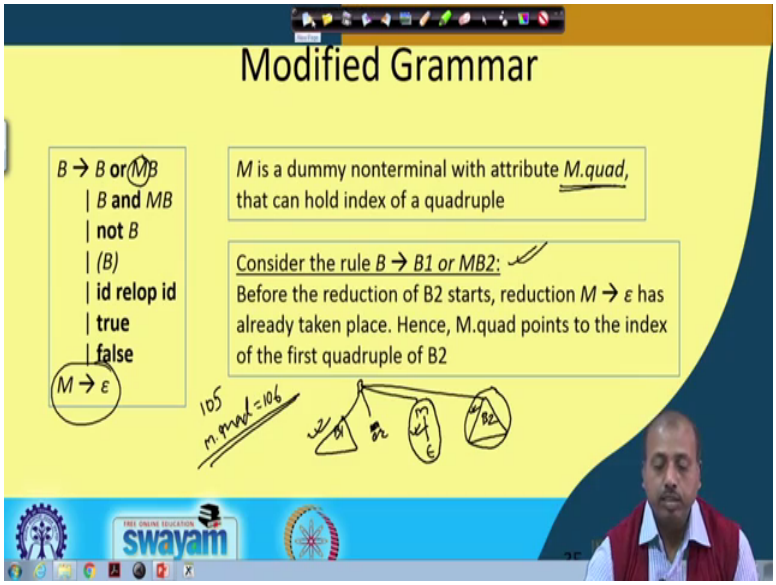
Some more functions are also necessary it makelist i. So, it creates a new list with a single entry i. So, and so, this is an index into the array of quadruples. So, some slide

earlier we are said that generated code is visualized as an array of quadruples, ok. So, here also we say that this makelist i.

So, if this is an index into that array of quadruples then mergelist of list 1 and list 2 it returns a new list containing list one followed by list 2. So, the margin of two list then backpatch list target. So, this will insert the target as the target label into each quadruple pointed to by entries in the list.

So, this will be. So, this list is containing all the places where this jump targets are not yet filled up, ok. Now for them so, if we have if we come across the correct target with which they to which these all these control should jump to. So, they that is say that that is called target. So, all those locations they will be filled up with the value of target and there is a function called nextquad that will return the index of next quadruple to be generated. So, many a time will need to know what is the address of the next piece of code that will be generated so, the at the next quadruple of code that will be generated. So, that is by this nextquad function.

(Refer Slide Time: 18:53)



So, the grammar that will have now is a modified version B producing B or M B, B producing B and M B not of B within bracket B i d relop i d true false and epsilon M producing epsilon. So, this is a. So, in this particular case, you see what has been done we have introduced a new dummy variable M, ok. So, we have introduced a new dummy variable M.

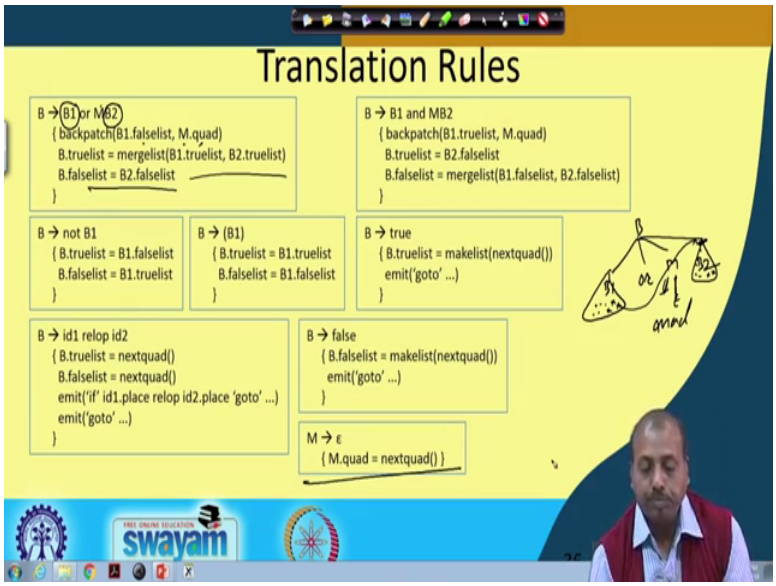
So, which we call and the this is this is dummy, because ultimately I am replacing it by M producing epsilon. If it does not have any effect on the on the M, ok, it does not have any effect on the grammar, but this will be useful for knowing some addresses and modifying the proper values inserting proper jump targets and all.

So, this M it has got an attribute called M dot quad that can hold index of the of a quadruple. So, that so, you can say M dot quad equal to some quadruple value. So, that will tell us this that will that will hold the in that can be a assigned some index of the of a quadruple.

Now, how is it going to be used? So, it can be it may be cleared by looking into say this particular rule B producing B 1 or M B 2, ok. So, before this execution of B 2 starts reduction M producing epsilon has already taken place. So, if you look into the parse tree. So, this will be like this. So, B producing this B 1 then M, sorry B 1 or then M and this B 2 and this M produces epsilon. So, this is the parse tree part. Now as we have seen previously by that number the policy of the numbering of those reductions. So, first this reductions will be numbered then this reduction will be numbered and then only this reductions will be numbered. So, before this B 2 reductions are made B 2 reductions are number B 2 reduction takes place just before that this M producing epsilon. So, this reduction will take place.

So, since this M producing epsilon is not going to give me any new code. So, this M dot quad which is the address of the. So, I can make that it can hold the index of a quadruple. So, if there is a function called nextquad. So, that gives me the index of the next quadruple to be generated. So, far we are generated till say up to this point, we are generated say 105 quadruples. So, this next quadruple is 106. So, what we can do is that we can make this M dot quad equal to 106. So, that will mean that that will be exploited later for correcting some addresses in the table. So, how is it being done? So, let us see in the successive slides.

(Refer Slide Time: 22:13)



So, this is the full set of rule that we have. So, we have got this first rule is B 1 B producing B 1 or M B 2. So, then what we do? So, at this point so, B 1 and B 2 they are truelist and falselists, they are known and this M it has got the quad as I was telling that at this point. So, I have got see this B 1 or M B 2. So, this was the situation and B 1 truelist has got all these points B 1 falselist has got this point B 2 truelist has got this point and B 2 falselist has got this first point say. So, this truelist and falselist have been done.

Now, with so, what will happen if B 1 is false? So, if B 1 is false. So, I have to come to the B 2 part. So, B 2 evaluations starts at portion when B 1 evaluation ends and B 2 evaluations starts at this point and that is that will be. So, before that this M producing epsilon. So, this will be done and for this M producing epsilon the corresponding action that we take is M dot quad equal to nextquad. So, this M has got a special attribute quad, ok.

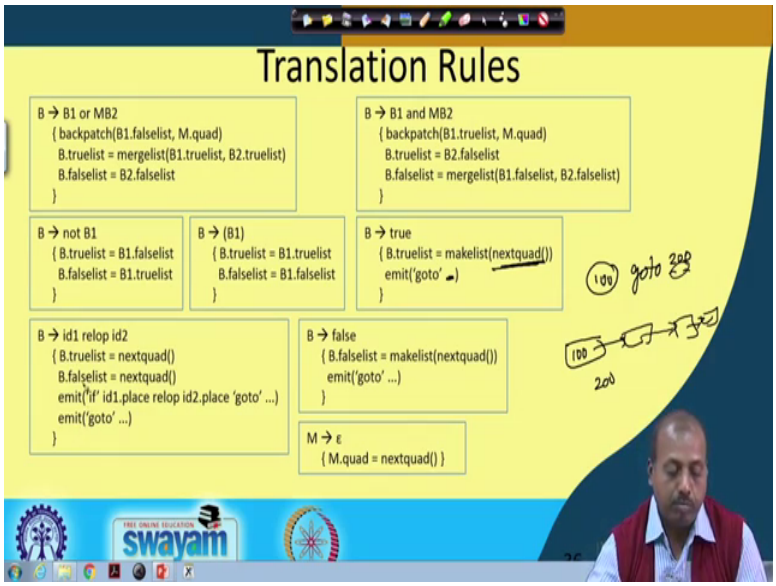
So, which holds the index of the next quadruple to be generated in the code generation process. So, this M dot quad is next quad. So, in this particular case. So, it will hold the start address of computation of B 2. So, this way so, this so, this backpatch B 1 falselist with M dot quad. So, all these falselist, all these positions they were they are target was targets were not known now. So, once this M dot quad is known. So, all those values should be backpatched with M dot quad.

So, all those blank locations they can be filled up with the value of M dot quad then B dot truelist. Soit is a. So, for B dot for B B truelist is the collection of all true points of B 1 and all true points of B 2. So, this is the truelist of B 1 of B say mergelist B 1 truelist and B 2 truelist. So, mergelist means, it will merge the truelists and get the overall thing and B falselist is B 2 dot falselist, because B ones falselist have already been backpatched with start address of B 2. So, it will branching from here to here in the execution and then B 2s falselist. So, that will be that will be. So, Bs falselist will be same as B 2s falselist similar type of rule can be made for this and so, B 1 and M B 2.

So, with backpatch B 1 truelist with M dot quad B truelist is made equal to B 2 dot falselist and B falselist is made mergelist of B 1 falselist and B 2 falselist. So, this is just the complementary version of this or rule the and rule now not of B 1 B producing not of B 1. So, B dot truelist equal to B 1 dot falselist and B dot falselist equal to B 1 dot truelist similarly B producing within bracket B 1. So, this is truelist and falselist remain unchanged B producing true. So, this will be this will have this will this is generate a code. And this at that point the expression is definitely true; because the expression itself is being reduced by the rule B producing true, true being the constant token for this is the constant value for the Boolean expression true.

So, that is this emit goto. So, this has to be filled up, ok. So, where to go we do not know, because once this target will be known then only this can be filled up. So, this way it is left like this, but before that this nextquad function has been called. So, this is a point where the expression is definitely true so, that so, sorry this, whenever this goto is generated.

(Refer Slide Time: 26:37)

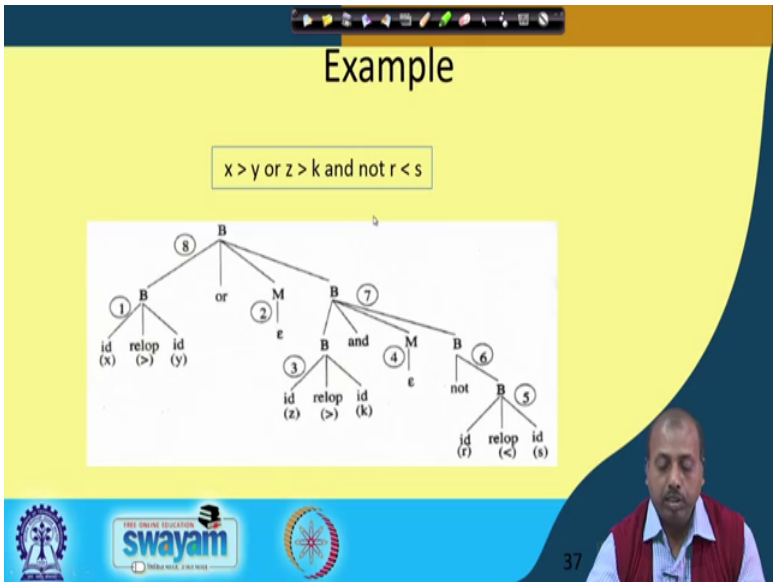


So, this particular quad, if it is quad number say 100. So, this is a place where I have to do the correction, ok. So, this makelist nextquad. So, it will make a list with 100 as one of the entry. Now there may be several other points that will come where B is definitely true. So, all these will be put into this chain, ok. And there will be that will be called B dot truelist and there will be backpatched with the when this goto target will be known.

So, all these places will be filled up since its target becomes a 200. So, I will be writing 200 here in that case. So, the code will be corrected then i d 1 relop i d 2. So, here also I have got the situation like B dot truelist equal to nextquad B dot falselist equal to nextquad then emit if i d 1 dot place relational operation i d 2 dot place goto and then again this part is not known.

So, this will filled up later when this B dot truelist this B dot true will become available. So, they will be backpatched with that, ok. So, this way we can have this Boolean grammar rules for them we can have the corresponding actions.

(Refer Slide Time: 28:05)



Now, so we can will look into this example. So, this x greater than y or z greater than k and not of r less than s, and this is the corresponding grammar that we have corresponding parse tree that we have. So, we initially have this B and since or is of list precedence. So, it is divided by this rule B producing B or B M B 2 M B then after that. So, this or is i d relop i d. So, this first B is i d relop i d. So, this is x greater than y then the second one is now this M, M is epsilon. Now, so, this now this so so, now, this so, this since or is of the list precedence. So, this is the second part of this or this whole thing is the second part this or and it has again had got two things like and not and etcetera.

So, first this before the end so, this z greater than k. So, that part is generated. So, B and M B that is generated and this from this B, we get i d relop i d and that is z and k M giving epsilon and then this B is gives me not of B. So, not of B and B giving me i d relop i d. So, r less than s so, this way we can generate the, we can write down the parse tree for this particular example string using the Boolean grammar. So, we look into the code generation for this in the next class.