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

Lecture – 50
Intermediate Code Generation

So, in next part of our course will be looking into Intermediate Code Generation. So, after so, far we have seen techniques by which we can just tell whether a program is syntactically correct or not and we can also do some amount of semantic checks like if the if all the symbols are defined properly; whether all the types have been defined properly; whether this program has got some type error ok; so all those things we have seen. Like then, we have also in the last chapter we have seen like how to what are the responsibilities for this management of runtime environment. So, what the compiler should do to do that.

But the major task that a compiler does apart from this checking of the syntax and all is to say is to generate the code which will be finally executed by the processor. And I have said that like a there is a phase before this called intermediate code generation which actually is not a mandatory one but it may very often so this forms a part of this compilation process because what can happen is that today maybe you are generating code, you are general writing a compiler that will generate code for say processor 1. Tomorrow you want to write a compiler for the same language, but it is targeted to a different processor; processor 2.

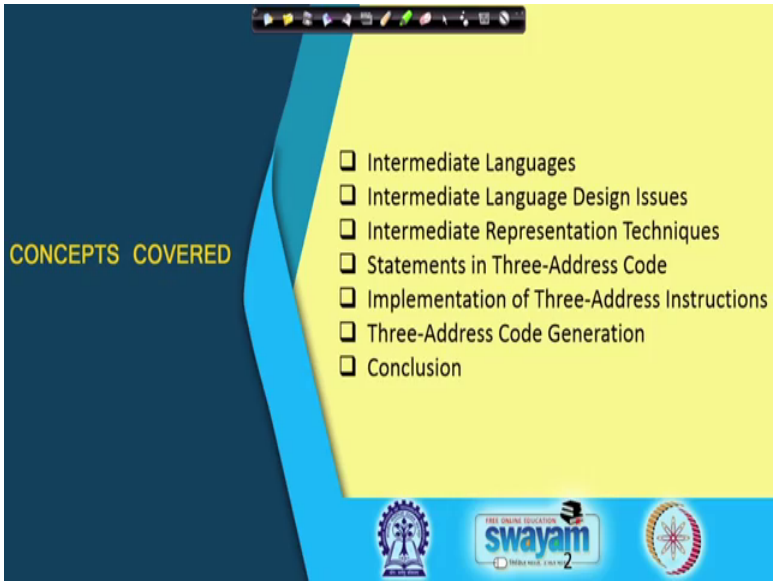
Now since the machine language of this processor 1 and processor 2 are different from each other. So, the code that you generated for processor 1 is not sufficient is not correct for processor 2 and there may be certain architectural features that are available in this processors such that if you can exploit those features the program will be more optimized.

So, that way we should try to generate we should try to have those things exploited. Now, if we concentrate on this entire compilation process for this machine 2 also from the very beginning, then the total amount of effort needed is just doubled ok. So, we so to make the situation simple; so, what is done is that normally we produce this code of this of a of a source language program up in some intermediary language and later on, from

this intermediary language the statements are converted into the target machine; so that is called the target code.

So, we have got the source program which we call source code that will be converted into an intermediate code and from the intermediate code it will be converted into this machine code or the target code. So, that is why this intermediary code generation often forms a very important part in this compilation process. And most of this language complexities are taken care of by this intermediary code generation phase such that when I go to the optimization phase so you do not need to be bothered much about the complicacies of the source language, syntax and semantics ok. So, they are already translated in some form and from that form we try to optimize.

(Refer Slide Time: 00:32)



So, this is what we are going to do in this particular chapter. So, as I said that this is an intermediary code or intermediate code. So, that has to have some language in which you write it. So, apparently it is a bit counter into it because he started with some source language program and then, we are converting it into some intermediary language program ok. But that will help us because this is intermediary language is going to be much simpler in their syntax compared to this source language that you are considering, so for which we are writing the compiler.

So, that way intermediary language is there going to be helpful. So, whenever I say so, so that is a language design issues, we will turn up like what are the types of statements

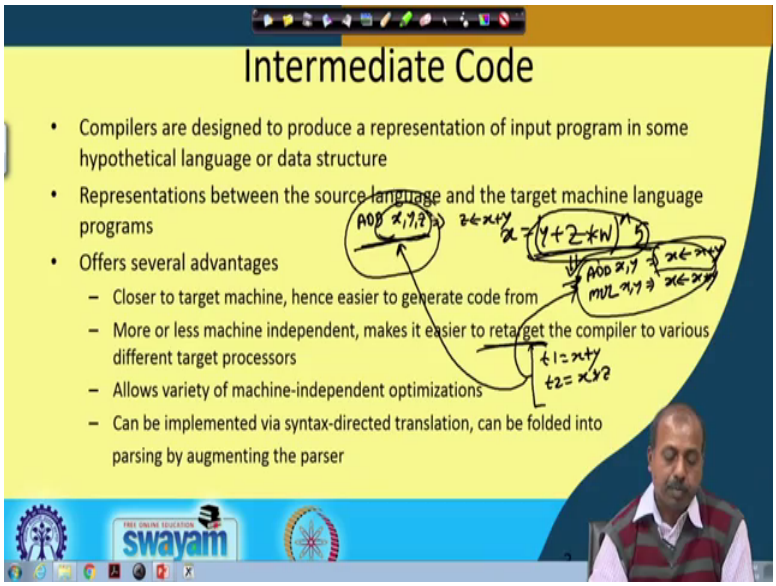
that I should have there; what we will be the complicacy of expressions that we have; do you have a loop there; do you have a this case statement or so, which case statement or do you have if then else or do you have function calls do you have pointers? So, all these issues we will come up.

So, we have to address these things like if you are say if you say that we will be generating target code in some language, then we have to some intermediate language. Then, what is the structure of that intermediate language that has to be told. So, this intermediate ah language representation techniques that also has to be thought about. Because ultimately so it may not be a source language program the text file this intermediate code may not be a text file. So, it may be that we if represent it in some other format, so that this representation becomes more efficient.

So, what are the statements that we are going to allow in Three-Address Code; Implementation of Three-Address Instruction like how are you going to implement it. So, that is the actually a part of the target code generation and this Three-Address Code generation process, how do you really generate code corresponding to this machinery this source language program statement. So, how do you really convert it into three address code statements so that is very important and then, we will be end up with a conclusion.

So, you see they this chapter forms the major part in the compilation process. So, far we were just trying to tell whether the program is syntactically correct or not. In some sense we were just passing the query of how to generate code. So, that we were just by passing. So, this chapter we will answer all your queries on that particular issue like how to generate the code.

(Refer Slide Time: 06:07)



So, to start with what is an Intermediate Code? So, compilers are designed to produce representation of input program in some hypothetical language or data structures. So, this is the hypothetical language. So, this is intermediary code; so this is written in some hypothetical language. So, these we will see that there are there can be several hypothetical language is that you can thought about that you can think about. So, then how to how to do this thing; so, how to generate this hypothetical language program; so that will be seen.

So, then the representations between the source language and target machine language program so, so how do you represent them? Then, there are several advantages. So, what are the advantages? So, it is closer to target machine and hence easier to generate code from. For example, it very simple case may be like this that in your source language, you may write a complex expression like x equal to y plus z into w to the power 5 like that. Now it may so happen that while in where the target machine on to which you are doing this implementation, it has got add instruction, it has got multiply instruction it has got so like that.

So, it has got only these two instructions and then, this add instruction it has got only say ah it can accept only say two operands x and y such that after doing this x, we will get x plus y. Similarly, this multiply instruction it has only 2 operands x and y and after executing this, it will be getting something like this; x equal to x into y. Now, see this

instruction is going to be very complex as for as this basic add and multiplication instruction is concerned; but in three in this intermediary language code what we will do? We will not allow such complex expressions. So, we will allow expressions like say t 1 equal to x plus y, t 2 equal to x into z; so like that we will allow such that is.

So, the see here the operands that we have they are very close to the target language operands and this the number of operators that we have they are close to target language operators and this number of operands are almost same as the target language statement. So, whatever operands are how many as many operands are there. So, here also for the operators we will have similar number of operands. So, that way it is made most or most they it is 2 operand; sometimes, it is 3 operand etcetera. So, it is closer to the target machine and hence, it is easier to generate code from. Second important point that we have it is more or less machine independent.

So, this statements like say t 1 equal to x plus y or t 2 equal to x into z. So, it does not assume any underlying architecture for the machine. So, when I say add mul this sort of statement. So, it immediately assumes that the underlying processor, it has got instructions like add, mul etcetera. So, but here I am not assuming anything like this. What is the exact syntax what is what is the exact machine language statement for doing this addition; so that is not important here. So, you are representing this operation in some hypothetical language and writing it has only with the restriction that arithmetic operation, it will have only 2 operands ok; so that way we are writing it.

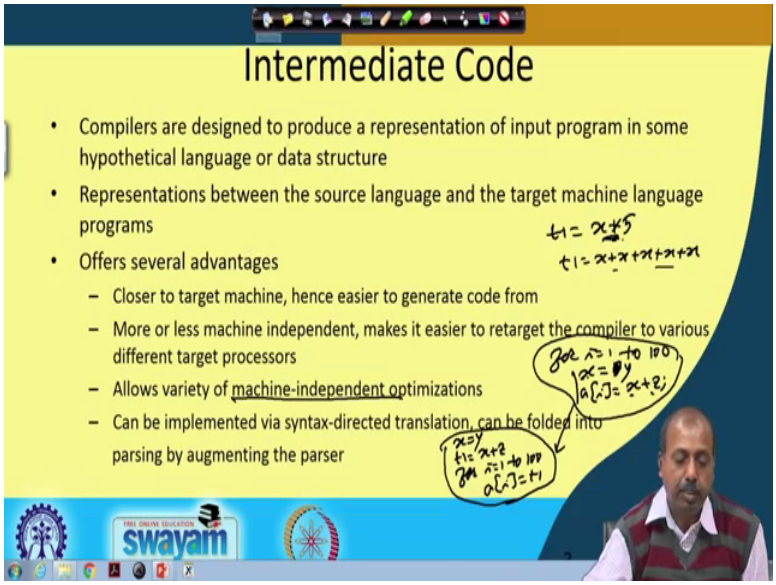
So, it is more or less machine-independent and thus, makes it easier to retarget the compiler to various different target processes. Otherwise what will happen is that there are so, if you look into this addition instruction itself, some processors you will find that it will so it will allow you to have 3 operand of format like add x, y, z such that the meaning is that z will get x plus y and some processors, they will allow you in this format add x, y telling that the operation is x equal to x plus y. So, in some cases it is a 3 operand instructions; some cases it is 2 operand instructions.

So, if you have generated code for a machine that supports 2 operand ah instruction and from there and now you want to modify the compiler so that it will be doing 3 operand. So, at it will support 3 operand instructions like this. So, they lot of changes are to be made, but you see that if you are doing it like this, then from here you can for the two

address you can generate like this and from three address you can generate like this. So, both are both can be done very easily; so that is why it makes it easier to retarget. So, this is called retargeting the compiler.

So, first the compiler was generating code for this machine and now it has been retargeted to generate code for this type of machines. So, that way there are many advantages. Then, it allows variety of machine-independent optimizations; machine independent optimizations are like this.

(Refer Slide Time: 11:18)



Suppose, I have got say ah say one operation say t 1 equal to x into 5. Now, it means that I will be multiplying x by 5 here. So, another possibility or doing it is t 1 equal to x plus x plus x plus x plus x. So, in most in many processors what happens is that this x this multiplication takes lot of time compare to addition. So, if I break it up like this, then it will be easier to handle. Then, there are many other optimizations. So, that way like it may so happen that suppose, I am writing a loop like this for i equal to 1 to 100; x equal to 5 and then so, say x equal to y and then, I am writing a i equal to x plus z.

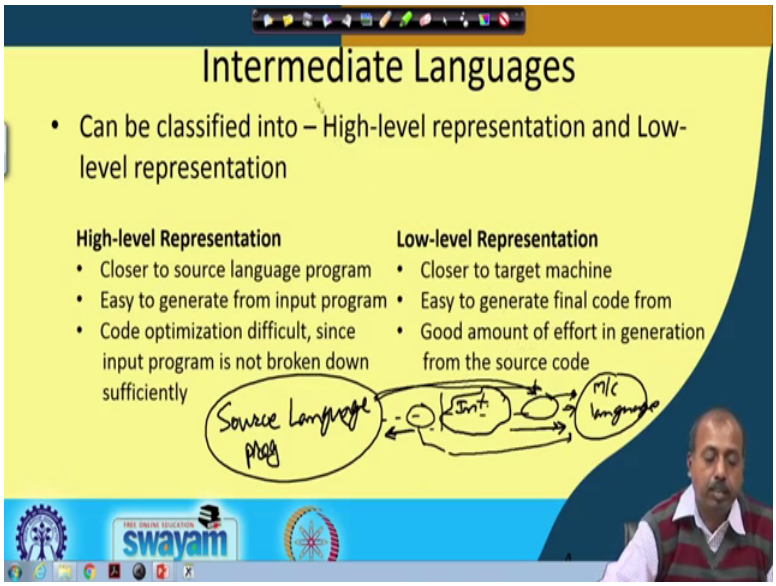
So, if I am writing like this, then you see this computation of this x plus x equal to y. So, assignment of y to x and then, this computation x plus z; so, this is also going to repeat 100 times. A better code can be write this like I write like a x equal to y and then, I take a temporary variable t 1 and mate make it equal to x plus z and then, I put this loop like for i equal to 1 to 100 a i equal to t 1. Why is it good? Because this computations, I am not

doing this 100 times. So, if I can do this thing in some intermediary language so that is independent of the target machine. So, this is the machine-independent optimization that I am talking about. So, this type of optimizations can be carried out easily if we in the three-address code it is a in the intermediate code itself.

And it can be implemented via syntax directed translation mechanism and thus, it can be folded into the parsing by augmenting the parser. So, this is the other advantage like we have seen that this type checking can be done in the using syntax directed translation mechanism. So, here this intermediate code generation also can be done by using the syntax directed translation mechanism and as we are doing this. So, this code generation becomes simpler. This intermediate code generation process become simpler and we do not need a separate phase further; so it is integrated with the parser.

When the parser we will particularly the shift reduce parser. So, when they do this reduction step. So, at the time or doing the reduction, so we can give it some set of rules. So, that it will follow those rules and as a result, the intermediate code we will get generated. So, these are the advantages that we have with this intermediary this intermediary language programs; so we are going to use them.

(Refer Slide Time: 14:11)



So, what can be the type of Intermediate Languages? So, there can be different class different types of intermediate languages that have been reported in the literature.

So, they can be broadly classified into High-level representation and Low-level representation. High-level representations; so, they are closer to the source language. So, what we are trying to do is you can understand that we have got this source language; we have got this source language program and that we are translating to machine language program. So, this we are taking to machine language and we have said that intermediary language in program is somewhere here; so this is the intermediary. Now, you can make this intermediary language close to this source language or you can make it close to the target language.

So, if you make to close to the source language, so we call it a high-level representation and similarly, if you make it close to the machine-language. Then, we will call it a low-level representation. Now, which one is better? So, it is difficult to answer. So, if it is close to this source language program, then it may be very easy that to generate this intermediary code because the language constructs may be more or less similar. So, this is the amount of job needed for doing the translation maybe small. However, so this part will become difficult then. So, if this intermediary language is here ok, it is closer to the source language; then, this translation is going to take more time.

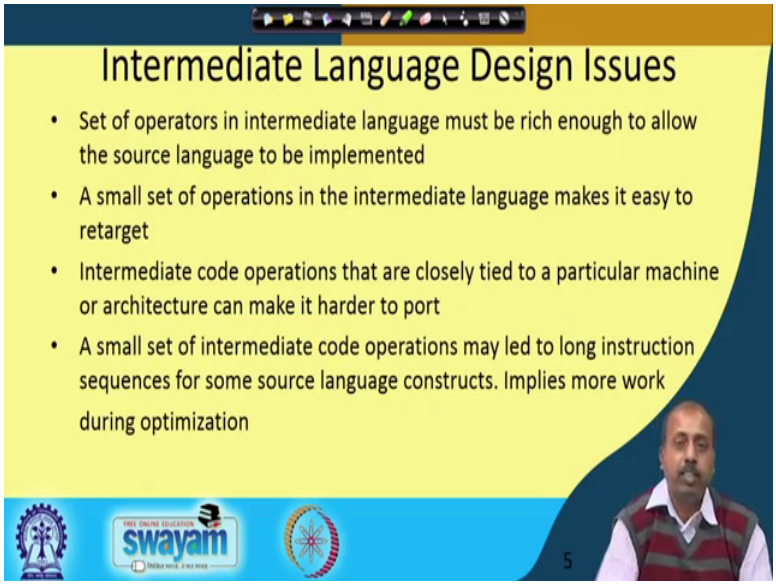
Similarly, if you take this intermediary language somewhere here, then this translation becomes may become very easy. There maybe one to one correspondence between this intermediary language statements and this machine language statements. But doing this part becomes difficult ok; it may become difficult. So, it is a very judicious choice like how much complex I should make my intermediary language representation; so that is a very judicious choice.

And high-level representations, so they are closer to source language program; they are easy to generate from input program, but the code optimization is difficult because the we are we are not very close to the machine language. So, many of the machine dependent optimizations; so they will they cannot be tried at this level and the input program is not broken down sufficiently. So, it we may not be able to apply many optimizations on that.

On the other hand this low level representation. So, they are closer to the target machine ok. They are easy to generate final code from. So, we can where as I was telling. So, there may be one to one correspondence at this point. So, that way this target code

generation may be very easy. But good amount of effort will be needed in this process from source program to this intermediary program. So, it may take good amount of effort. So, this way we can have different types of intermediate languages and they may have different advantages and disadvantages also. Now, how are you going to handle this situation like what are the different intermediary languages that have been proposed.

(Refer Slide Time: 17:28)



And before that we try to see what are the issues in the in such language design. First of all this set of operators in intermediate language must be reach enough to allow the source language to be implemented; so this is the first thing. So, may be in my source language there is an exponentiation operator. Now, I should keep an exponentiation operator in the intermediate language also.

Otherwise, what will happen is that I will try to implement that exponentiation operation by means of multiplication in the intermediate language only to find that in the target processor already and exponentiation operator is available. So, again there will be reconversion from this chain of multiplications to exponentiation or many cases what happens is that this subtraction operation may not be supported or so if the subtraction is not supported, then instead of that maybe I can multiply with minus 1.

So, I would multiply with minus 1 and then add. Now, in the intermediary language; so if subtraction is not directly support it, then I may do like that only to find that in the target processor subtraction instruction is available. So, it is desirable that the set of operators

of intermediate language must be reach enough to allow the source language to be implemented; otherwise it becomes so difficult like if it is not possible to do something that is possible in the source language, but not in intermediate language; then, we cannot carry forward it. For example, maybe my source language is allowing me to or do of real number arithmetic, but my intermediary intermediate language does not allow.

So, naturally this real arithmetic cannot be taken forward to the next level; so that this type of problems can be there. So, this language the intermediate language that we are suggesting will not be applicable for this particular case. At similarly at this a small set of operations in the intermediate language makes it easy to retarget. So, this is the other issue like just being afraid that if I do not include source language operators in to my intermediate language. So, it may make it you less useful. So, we may incorporate many of this operators onto this intermediate language. But there then, then for each of them, we have need to have the corresponding translation into machine code the target for the target processor.

So, the retargeting becomes difficult in that situation. So, a small set of operations in the intermediate language makes it easy to retarget. So, if there are large number of operations; so that we will also make a difficult. Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port. Like there may be certain operations which are specific for a particular processor.

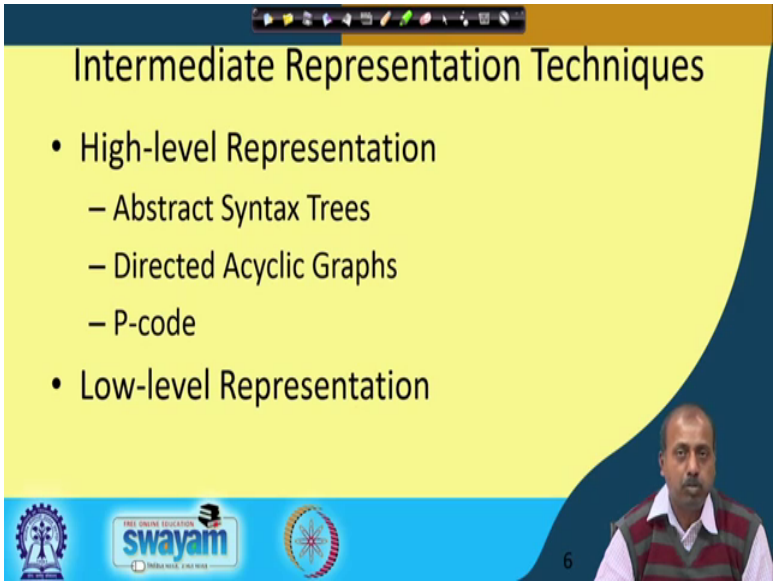
For example, in case of say hm say this X-86 architecture there is an instruction by which you can copy a block of memory locations from one region of memory to another region. So, that is the very useful instruction, very useful feature; but if we do if we do use that in case of say array copy or something like that in the intermediate language, then a processor which does not support it; so there it will become difficult.

So, we should not be very tightly guided by some target processor architecture and instruction set to decide upon the three this intermediate language instructions. On the other hand, a small set of intermediate code operations may lead to long instruction sequence for some source language constructs. So, that can also happen. Like if the if my instructions are very simple in case of intermediate language, then some source language program when we are trying to write in terms of intermediate language; then it may

require very large code. So, that can happen; so that has to be taken care of. So, it implies more work during optimization. So, more work will be necessary during optimization.

Because the code at this point itself become sub optimal and then, we are code optimization phase which is anywhere very costly; so, that is going to take more amount of time. So, these are the issues that we have with the so with this intermediate language design problem and we have to do something, we have to select a language for this intermediate representation; so that the issues are taken care off.

(Refer Slide Time: 22:07)



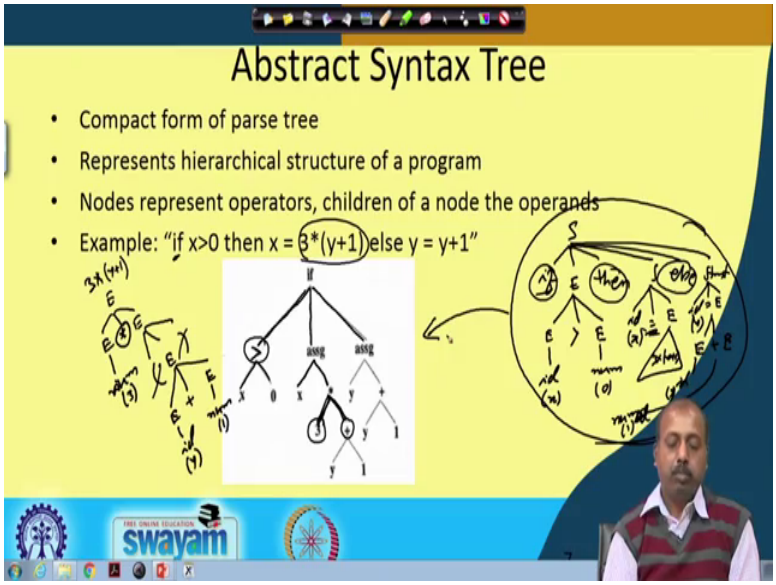
So, Intermediate Representation Techniques. So, we have got High-level Representations like Abstract Syntax Tree. Then, Directed Acyclic Graphs and something called P-code and there are some Low-level Representation techniques. So, we will see slowly like what are these methods.

So, this Abstract Syntax Trees; so, this is in some sense this is a representation of the parse tree only. So, this is the syntax tree. So, parse tree in some sense, so it represents the it is it is some sense it is represents the program. So, if you just take this parse tree say this terminals that are coming and convert them into some other form which may be bit different from the source language; so that may give rise to a syntax tree.

Then Directed Acyclic Graph; so this is for between the operators, we can between the operands we have got certain operators. So, we can represent them in the form of a DAG

and we can happy code so that we will see. And the Low-level representation, so they are ah very close to the machine language; so, we will see what are those things.

(Refer Slide Time: 23:23)



So, this is the Abstract Syntax Tree type of representation. The compact form of parse tree as I was telling that parse tree in some sense, so it represents the program. So, you can exploit that particular structure for representing some representing the program in a different form. So, this is a compact form of parse tree and represents hierarchical structure of a program. Nodes represent operators children of a node the operands like if I have got something like this that if x greater than 0; then x equal to 3 into y plus 1 else y equal to y plus 1.

So, we can say that so the parse tree for this depending on the programming language, so it will be some statement giving rise to if expression, then statement and this else statement and then, this E giving rise to so relational operator E greater then another relational operator and then these E giving raise to id and that is your x and this E giving rise to number that is the 0.

Similarly, this S giving raise to id equal to some expression and this expression somehow giving this 3 into y plus 1. This statement giving rise to again the statement that id equal to E; This id is y. This id was x and then, this is giving E plus E etcetera. So, you giving E plus E and then ultimately this E giving id; this E also giving id. So, this id this giving number which is 1 and this is y.

Now, you see. So, this is the syntax tree. So, the parser has generated this thing. So, you can just use a slightly different format from here, where we say that this if statement. So, we take the if statement and we represent in the abstract syntax tree since it has got 3 portions; one is the condition, one is the then part assignment, one is the else part assignment. So, you do not; we do not store this tokens explicitly like if, then, else then etcetera. So, those are not there not relevant because here it means that this is the this is the link for the condition. This is the link for the then and this is the link for else; so knowing that I do not need to do anything.

Now, for the condition part, so this is a greater than. So, this has to be stored because there can be different conditions. So, I need to store the condition that we have. So, this is greater than and after greater than we have got this x and 0. So, they are necessarily. So, similarly this assignment; so assignment means if they are there will be 2 sides; 1 is the left side, 1 is the right side. So, I do not need to store this equality in the syntax tree. So, this is equality is not stored. So, rather this x is stored on the left side and the right side, I store this expression 3 into y plus 1; so this node is star.

So, since this is a binary operator, there will be 2 operands. So, I do not need any other thing. So, I just store three here and then this is plus. So, it is y plus 1. In fact, if you look into this parse tree for this 3 into y plus 1. So, it is not very simple because the way it will be generated is something like this; so E star E. Then, this E giving number which is 3 and then this E giving bracket start E bracket close and then, this will give us E plus E. This E giving us id which is y and this E giving us number which is 1. So, this is such a complex thing, but we do we really need to remember all this things. Because as soon as we know that this is star, so I should have this number this left and right side of this left and right operands of this multiplication operation.

So, that is done here; so, this 3. So, this is the left side of the operand the left operand these are these are right operand and similarly, for plus also; so, this bracket and all, so they are not necessary just removed. So, that way from the parse tree, you can do some modifications so that the non essential portions of the parse tree, they are just trimmed out and what you have got is the bare minimum skeleton which will be able to represent the instructions or the statements of the programming language. So, in this so this is a representation which is very close to the source program. At the same time it removes

many of the details of the source program and still it captures the essence of the program for which we are trying to generate the target code.