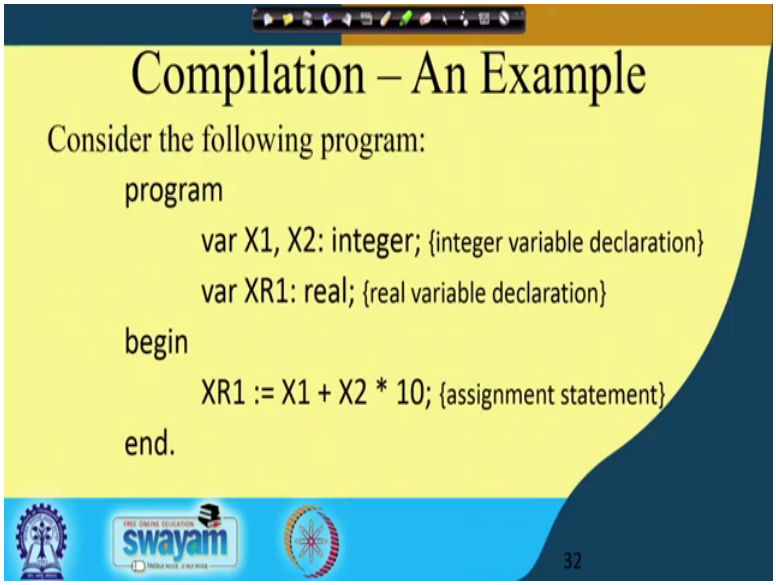
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 06
Introduction (Contd.)

Next will be looking into an example and with that example, we will try to explain this Compilation Process, how it process, how it goes, and what are the output of various stages, ok.

(Refer Slide Time: 00:25)



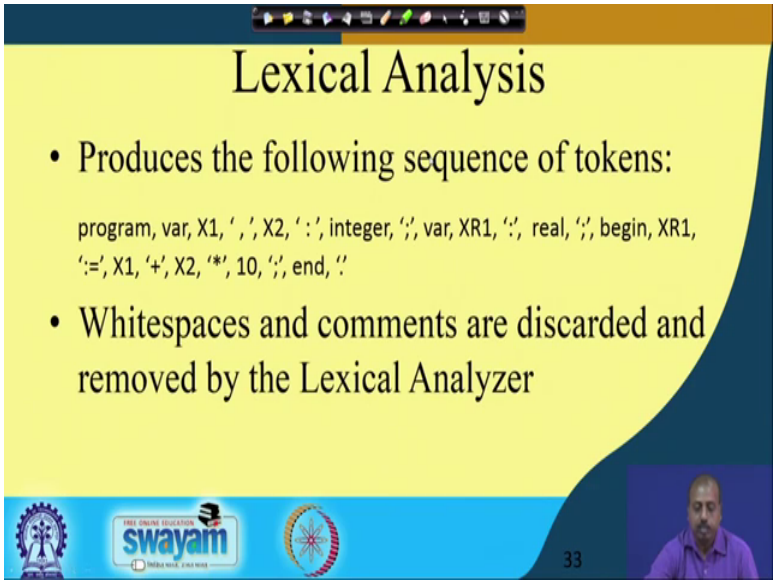
So, we take an example from of a program which is close to ah the language called Pascal, ok. So, anyway it does not matter. So, the essentially this programs in that language starts with the keyword program, ok. Then comes the variable declarations, and each variable declaration is preceded by the keyword var, ok. So, we have got var X1, X2 then colon integer it says that X1 and X2 they are two variables of type integer. Then we have got XR 1 as a real variable. So, you can have this var XR 1 real. So, these are the variable declaration.

So, once the variable declaration is over, so we can start the main part of the program like in see language we have got this main function. So, in Pascal type of, Pascal language that we have considering here, so after program this program keyword so

whenever you are given this begin keyword it means that we are going to start writing the main body. So, the execution we will start from this point onwards.

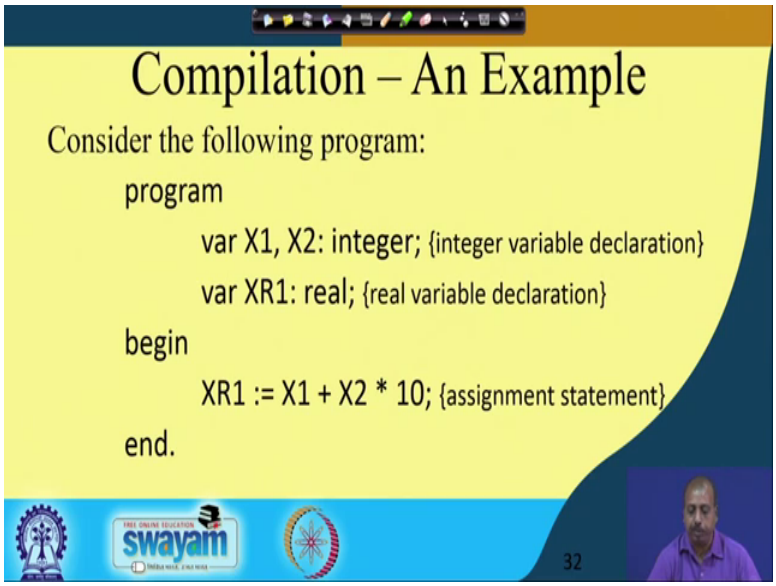
So, here in this main body, so we have got a single statement XR 1 assigned as. So, this is the assignment operator colon and equality. So, X1 plus X2 into 10. So, that is an assignment statement. And the program ends are this point, so this end is indentified by end dot. So, this is the full program. So, this is the very simple program that we have and that is the whole and that is the whole program that we have taken. So, we will see what are the outputs of various stages of the compiler as it processes this particular input.

(Refer Slide Time: 02:08)



So, first stage of the compiler is the lexical analysis. So, lexical analysis stage as I said that it will remove all this white spaces like this blanks, tabs, new line characters, then the comments.

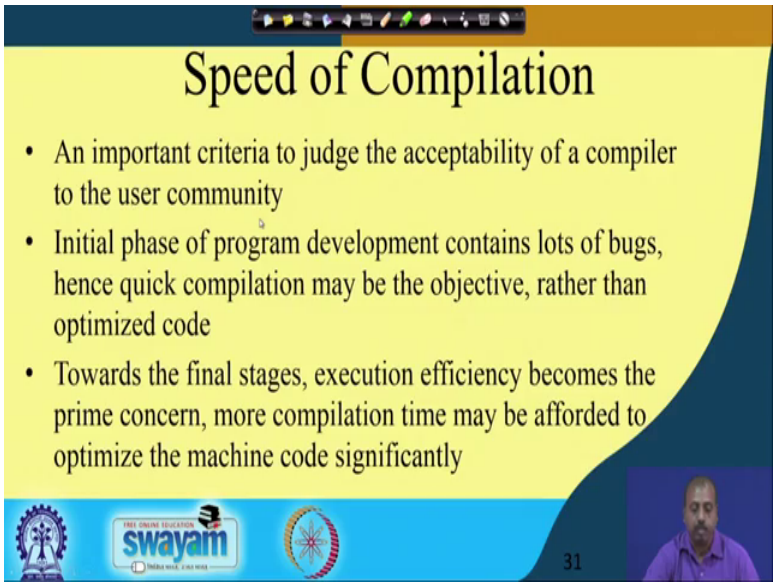
(Refer Slide Time: 02:23)



So, if you look here, so these are the comment. So, whatever we put within these curly brace. So, they are all comment. So, all these are removed. Similarly these new line characters, then these tabs, then these blanks, they will all be removed.

Ideally I can write this entire program in a single line, by just demarcating each word by a single blank space. But we can write in a fancy way also as we have done it here and the lexical analysis tool is suppose to if I remove all those cosmetic from the program.

(Refer Slide Time: 02:58)

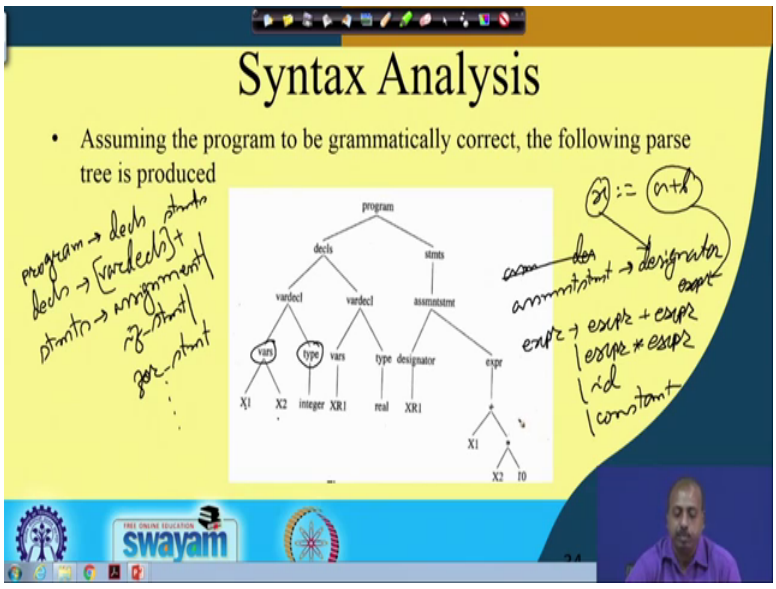


So, this is lexical analysis tool after analyzing the program. So, it produces a sequence of tokens. Like, so for the keyword program it outputs the token program. So, actually as I said that this, tokens are nothing, but some integer values. So, for the sake of our understanding we are not writing in terms of integer values, but we can assume that all these words like program, var, so they are having some corresponding integer value.

So, the lexical analyzer is actually passing those integers to the syntax analyzer or parser, but for our understanding we will write them as the this in terms of this tokens only program, var etcetera. So, this program token is returned. So, when it sees that this keyword program it returns the token program, then these were then X1, then comma, then these X2, then these colon, then these integer, then semi colon, var, XR 1, colon, real, colon, semi colon, begin, XR 1. So, these are the various tokens. So, when whenever the parser asked the lexical analysis tool to give the next token it gives one of these tokens which comes next, ok.

So, whitespaces and comments are discarded and removed by the lexical analyzer. So, that we have discussed previously that this all this whitespaces the cosmetics think that are removed from the program. So, at the output of the lexical analysis is nothing but this sequence of tokens, and the sequence that token is returned to the parser only when the parser asks for the token.

(Refer Slide Time: 04:44)



Now, coming to the syntax analysis phase. So, we will assume that the program, if we assume that the program is grammatically correct. So, it will generate a parse tree. So, I said that parse tree is a 3 type of representation that tells how the program can, how this entire program can be taken to, have been derived. So, this whole program is divide is this is actually depicted more by the grammar rule that we have in the program which you for the language which is not shown in our example, but assuming that I have a rule which says that the program can produce declarations and statements.

So, there is a rule which says that program is can produce declarations and statements. So, first whenever it sees the program keyword it divides it into portion declarations and statements. Similarly there is another rule in the language which says the declarations can be a number of variable declarations. So, we can have a number of variable declarations, ok. So, this is so when you come to this parser chapter. So, we will be discussing more on the grammar how to write it and all. So, it is like this.

Similarly the statements that we have statements can be of different types, like assignment is one statement. Then we can have say if-then-else, if statement is another state type of statement, then for statement is another type of statement. So, this way we can have different types of statement. So, that is; so we have got rule in the grammar which goes like this.

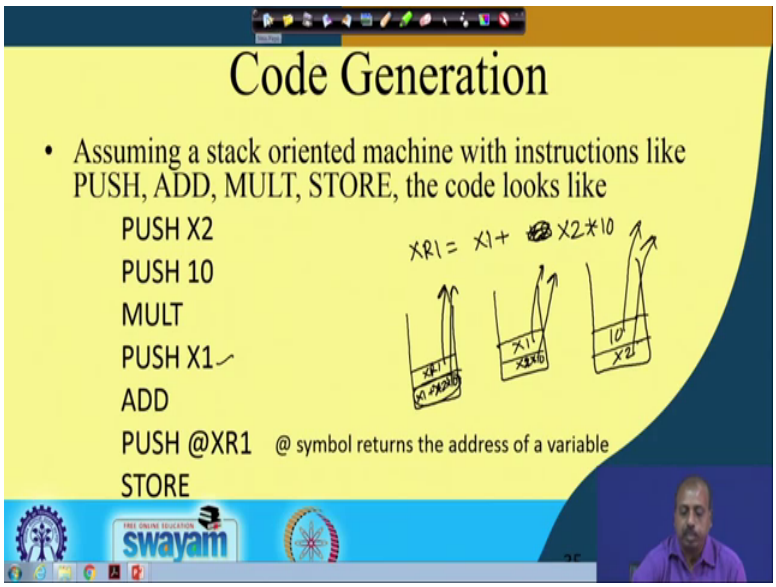
Then the one variable declaration it has got this is the list of variables. So, that is identified by vars and then the type. Similarly, this is the X1, X2, so this list of variables are identified by this is identifiers ended with a colon, ok. So, this that entire thing is retained here, so X1, X2, so integer. So, this way the the entire program that we have it can be represented in the form of a tree.

So, assignment statements, it has got a designator and expression. So, the thus designator the rule is like this that assignment assignment statement, so this assignment statement it has got a designator and expression. So, if I have got say X assign to some a plus b then this X is the designator part and this a plus b is the expression part. So, we have got an got an assignment to the variable XR 1. So, the XR 1 is the designator identifier and this is the expression that we have is X1 plus X2 into 10. So, this expression has got its own grammar, say expression is expression plus expression or expression star expression or it can be an identifier or it can be some constant.

So, here it is divided like this. This expression is expression plus expression like that, and then the right hand side is again expression multiplied by expression. Then the expression the light left hand side it gives me the identifier expression on the right hand side gives me the constant. So, this way we can have different we can we can generate the entire parse tree.

So, if the program is syntactically correct then this syntax analysis tool. So, this will generate a parse tree. So, this parse trees is used by the code generation stage for generating the generating code for the correct program. And if the program was syntactically wrong then you this parse tree could not be generated by using the rules of the grammar, and accordingly the parser will give some syntax errors and at the other compiler designer can flash appropriate syntax error messages for the program, for the input now. So, assuming that the program is syntactically correct, so it generates this particular program.

(Refer Slide Time: 09:24)



Now, the code generation. So, assuming that we are generating code for some stack based machine with the instructions like PUSH, ADD, multiplies, STORE etcetera the code will look something like this. First I need to evaluate this expression in the, so the code is to evaluate the expression that is XR 1 equal to X1 plus X2 sorry, X1 plus X2 into 10. So, it first pushes this X2 into the stack. Actually the way it operates is like this.

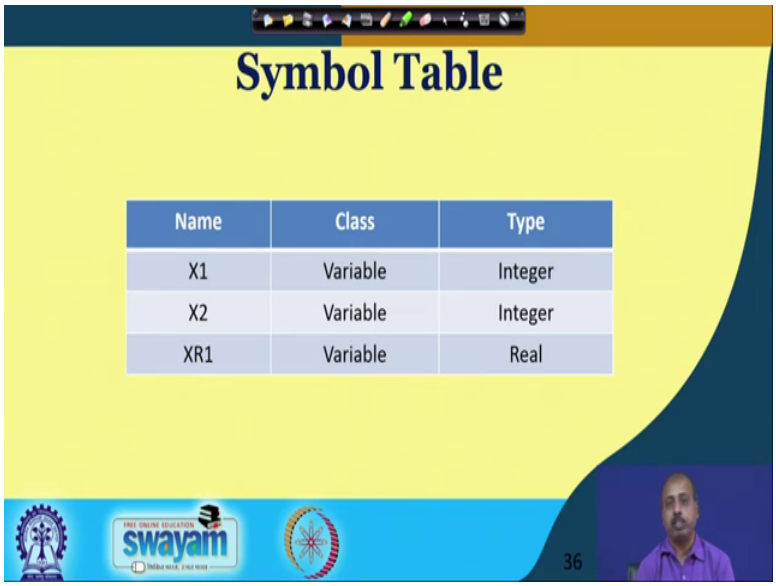
So, first it first it has to evaluate this X2 into 10, and for doing that it has to PUSH X2 and PUSH 10 and then call the multiply operation.

So, as the stack based machine, you know that it pops out the two top most interest from the stack. So, in the execution what will happen is that sorry in the execution what will happen is that we have got in the stack first X2 pushed and then 10 pushed then this is multiply instruction comes accordingly the machine while executing it, it will pop out this stain, and X2 from the stack, do the multiplication and PUSH the result back to the stack. So, now, after doing this the stack will after this PUSH statement has been executed the stack will have X1, sorry it will have the a value of X2 into 10.

Then it says PUSH X1, so X1 is pushed into the stack and then it does ADD. So, when it does ADD, so again these two are popped out, the values are added and then; return value also this value has to be so now, my stack has got this result. So, this is X1 plus X2 into 10, X1 plus X2 into 10. So, this value is there in the stack. Then in the stack we PUSH the address of XR 1. So, the address of XR 1 is pushed into the stack and then we say STORE. So, when we say STORE it will again pop out to top most entrees the first entry will give me the address of the location where you want to store the value and the second entry gives me the actual value. So, based on that, it will be, it will be storing the value of this expression on to the variable XR 1.

So, first stack based machine. So, it will generate code like this what for some other machines, it will generate code in a different fashion. So, that explains the compilation process. So, starting with the program lexical analyzer we will identify the tokens. Parser will arrange those tokens in some fashion, so that it becomes the derivable program from the start symbol of the grammar. And if it can make that parse tree then this code generation face it can generate code by taking help of the parse tree the individual stages of the parse tree it can use and generate the code for the target machine. So, this way this compiler works.

(Refer Slide Time: 12:26)

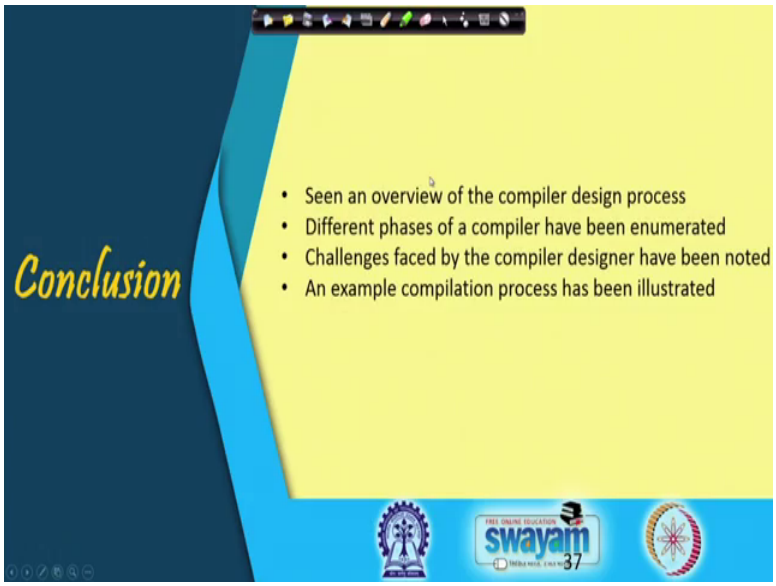


Then the symbol table as I said that there are three symbols in this program X1, X2 and XR 1. So, their class, the class is all of them are of type vary or of class variable X1 and X2 are of type integer and this XR 1 is of type real. So, that way we can have this X1, X2 and XR 1. So, there of time integer (Refer Time: 12:50) though.

Another field can be there which is the offset which is not shown here. I have not shown it explicable, purposefully because just to emphasize that it is not mandatory that all those information with their, all symbol tables be similar, so it can vary. So, of to the compiler designer, compiler designer may feel that certain information be there in the symbol table certain information may not be there. So, in this case the compiler designer might have thought that these are very simple. So, I do not need to, I do not need the offset. So, what we are doings for this stack based machine? So, we are pushing them on to the stack. So, as a result we do not need the offset values. So, this may be the symbol table structure.

Then after that we have got this code generation is done. So, the symbol table is done.

(Refer Slide Time: 13:35)



So, with this example we will like to conclude this part of our discussion. So, in this discussion we have seen an overview of the compiler design process. We have seen different phases of a compiler, how are they, what are they. So, we have enumerated the phases we have also discussed in detail the challenges faced by the compiler designer. So, as a successful compiler designer we have to take care of all those points, and we have also illustrated by means of an example compilation process.

So, with that I will thank you. And in the next class, we will go to the lexical analysis phase and start in more detail like how to design a lexical analyzer for a compiler.