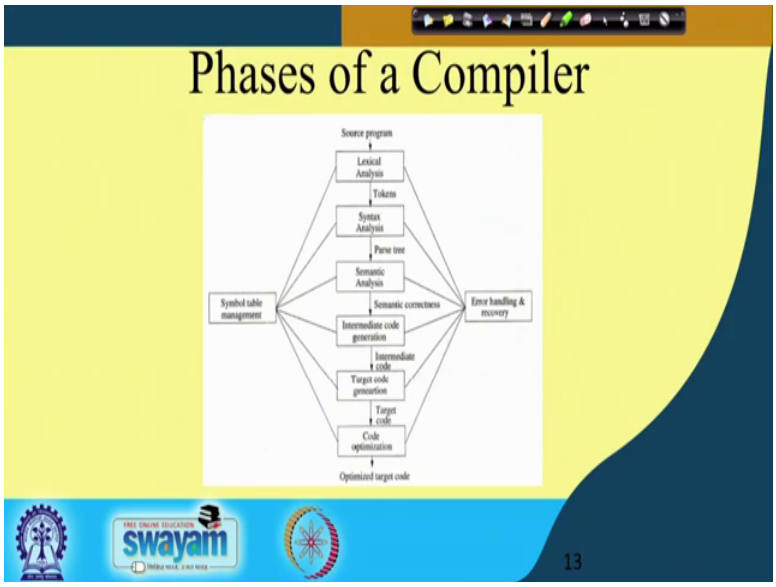
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 03
Introduction (Contd.)

So, we are discussing on phases of a compiler so in that we can see that it starts with the source program and that source program goes through a serious of transformations through various stages, and these stages they work hand in hand to produce the final optimized code.

(Refer Slide Time: 00:32)



So, to start with the source program goes through a phase called lexical analysis, so this lexical analysis phase it takes out, it identifies the words that you have in the program. Then so that formally they are called token because this is something more than the word so it is some identified corresponding to the words that we have plus some information like if it is a number; for example, then what is the value of that number. So, accordingly that the token may contain a token id part and one value parts.

So, we will see that later. Those tokens are used by the syntax analysis phase in the syntax analysis phase we check the grammatical correctness of the program. So the grammar rules of the language they are consulted. And whether the words appearing in a

particular sequence gives rise to a meaningful program or not or syntactically correct program or not so that is evaluated by the syntax analysis phase.

So, output of the syntax analysis phase is a part 3. So, that shows how the grammar rules of the program can be utilized to show that this program is syntactically correct or grammatically correct.

And so, grammatical correctness does not mean that the program is also semantically correct. For example, say one integer variable and the real variable both of them are identified as variable, but what we need is that we need to also sometimes we need the certain operations that can be done on integers certain operations can be done on real.

So, we need to categorize between these 2 types of variable. So, that those are done by the semantic analysis phase, and there we check whether the way the program constructs or are the components in the program had been used so whether that is correct or not. So if the syntax and semantics of the programmer correct, then it goes into the code generation process, and there it goes through a 2 phase code generation.

In the first phase one intermediary code is generated and from there the final target code is generated. So, intermediary code is in some hypothetical language and from there whichever processor for usually want to generate the code that is the target processor so for the target processor the code is generated.

So, the target code it goes through optimization because of these automated process they are remains many scopes at which the program can be or the generated code can be made more efficient in terms of execution speed, in terms of area or the total storage requirement. So, that way so it goes into code optimization phase.

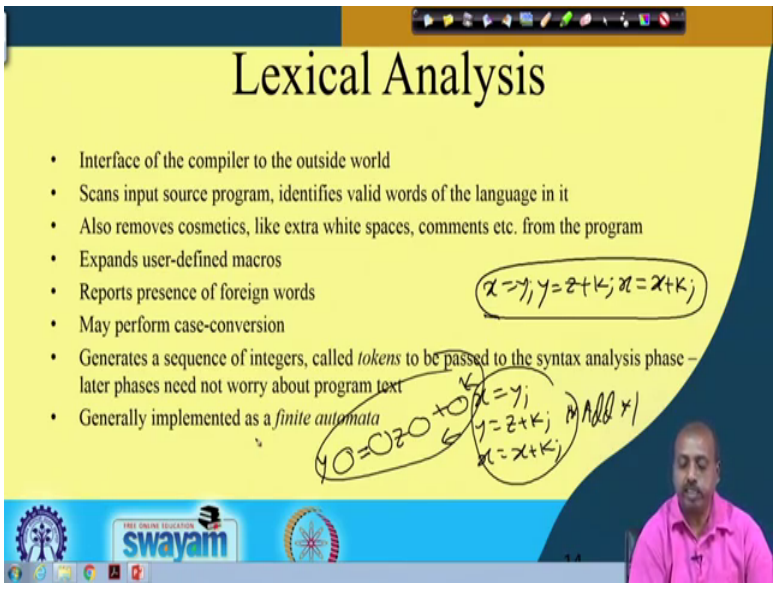
And output of the code optimization phase is optimized target code. Now as we have discussed in the previous classes that ok, so, it is not mandatory that the compiler will produce only optimized code so for silicon compilers it will produce optimize circuitry. So, they are also optimization is necessary because it may so happen for example, 2 inverters they come one after the other. So, an optimization may remove those 2 inverters because they replace them by a equivalent operations. So, that way so this type of optimizations are necessary.

So apart from this major flow so there are 2 more operations that are done in a compiler, one is the symbol table management so all the symbols that are defined in the program. They are kept in a table called symbol table. And this lexical analysis phase and syntax analysis phase they actually make the symbol table and the later phases they are actually going to use the symbol table.

And there is another module which is error handling and recovery of module. So, that actually tries to give hint to the user like what are the possible errors in the program, and accordingly it will the user will rectify the program and give it for compilation again. And while doing this error handling so it may so happened that the compiler itself goes into a state from which it cannot proceed further ok. So, in that case some recover reaction is necessary so that is why this stage is called error handling and recovery. So, recovery maybe it will discuss some of the last few words that it has been in the source program. So, that it can come to a valid state.

Typical example is maybe if there is some error in a line so if in the syntax of the programming language says that every statement should end with a semi colon, there it will skip all the symbols till it gets a semi colon because after that it knows that ok, I will be in a clean state where I expect a new sentence. So, that way this error handling and recovery routines are going to be useful.

(Refer Slide Time: 05:33)



So, we will see them in this part of the lecture we will try to have an overview. And later on for each of these phrases we will dedicate quite a few lectures discussing about their development. So, the first phase is the lexical analysis phase, so this is the interface of the compiler to the outside world so any program or the component that we want to compile. So, it comes to the lexical analysis phase. So, the major job of the lexical analysis phase is to scan the input source program and identify valid words of the language in it.

So, as I was telling that at the lowest level of a language you have got alphabet set. So, every language will accept certain alphabet, and those alphabets are combined in some fashion to make the valid words. Now, if those if the next combination of this alphabet is such that these does not make any word of the language, then the lexical analysis phase is supposed to identify that, and then it can tell that this word is not known to this particular language so this that is an error.

So, that is the purpose of the second point that we are discussing, it will scan the input source program and identify the valid words of the language unit. Now apparently it seems it is a very trivial job, but it is not so really because when we are writing programs so, we are we are very much flexible. Like in the sense that somebody some programmer may write the entire program in a single line because it just says that after going from one line to the another line you have to put a semi colon. So, somebody may write a program in this fashion say, x equal to y semi colon y equal to z plus k then x equal to x plus k so, something like this.

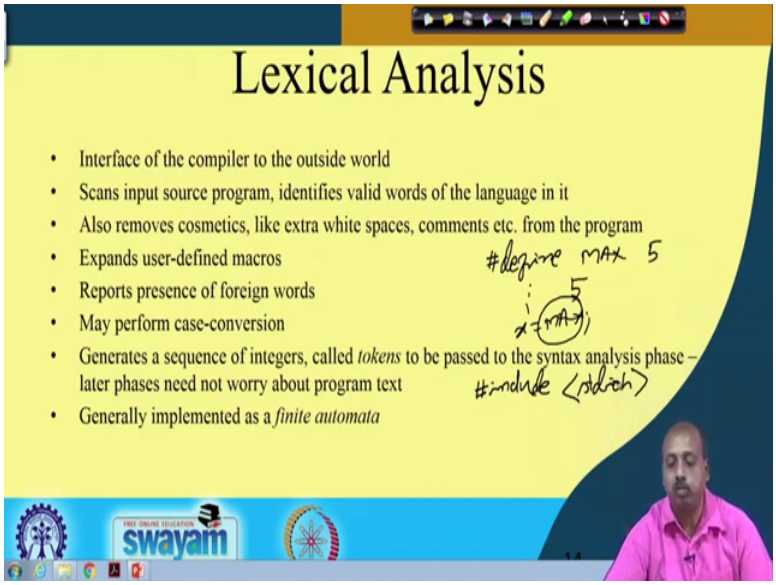
Somebody may put it in separate lines x equal to y in one line, then y equal to z plus k in another line, then x equal to x plus k in another line, somebody may like to put some comments also here, somebody may put a comment here at and this comments may also run in multiline fashion. And this somebody may write this line y is equal to z plus k as y then a quite a few blanks then z, then again quite a few blanks then k.

So, this type of variants variations in the source program can occur. So, we need to identify, you need to identify this situation somehow we need to ignore all this extra cosmetic things that are put into the line. Similarly, so whether the program is written in this fashion or this fashion or this fashion everything is correct ok, all of them are correct.

So, this lexical analysis phase it is responsible to take out the actual words meaning full words from the program so that is why it is job is a bit difficult. So, we will see that so it will remove cosmetics. So, as I said extra whitespace so whitespace means that blank, then tab, then new line characters.

So, they are all white spaces so it will try to remove it will remove all the extra white spaces comments, so you may put some comments in lines of comments are not compiled. So, they are not meaning full to the machine. So, they are to be removed. So, that removal is also done by the lexical analysis phase. So, these are some of the responsibilities, in many programming languages for example, in c language you will find that we have got macros like we say, hash define hash define say max 5.

(Refer Slide Time: 09:19)



And later on in the program wherever I write say x equal to max. So, that this max has to be replaced by 5. And that is we know that it is this translation or this transformation is done at the compile time. So, this max is replaced by 5 at the compile time.

So, this is also actually done by the lexical analysis phase. Similarly, we have got another compiler directive like hash include in c language. So, we say that hash include stdio dot h or some any other file.

So, what it means from the language we know that this stdio dot h this particular file will be attached to your program before compilation. So, before so this stdio dot h file has to

be taken and it has to be attached with your program. So these are all done by the lexical analysis phase.

So is expands the user defined macros, they has defines hash includes excreta so all those pre compiler direct this they are all expanded by the lexical analysis phase. Reports presents of foreign words, as I was telling that some words, some sequence of alphabets maybe put which is meaningless for the particular language.

So, it can report that such an such word is not in my language. So, that is the foreign word; may perform case conversion like sometimes the we need to do case conversion upper case to lowercase or lowercase to uppercase.

So, sometimes a language is case sensitive sometimes it is not. So, accordingly this lexical analysis phase it may have to do some case conversion. Now as an output, this lexical analysis phase it generates a sequence of integers called tokens. So, all the before this lexical analysis phase what we have is a program which is a character stream.

So, it is a stream of characters, but at the output of the lexical analysis phase, what the lexical analysis tool has done, is that it has identified the words that are present in those character stream. And for every word it has got a predefined integer. The compiler has a predefined integer value and that value is passed from that point onwards.

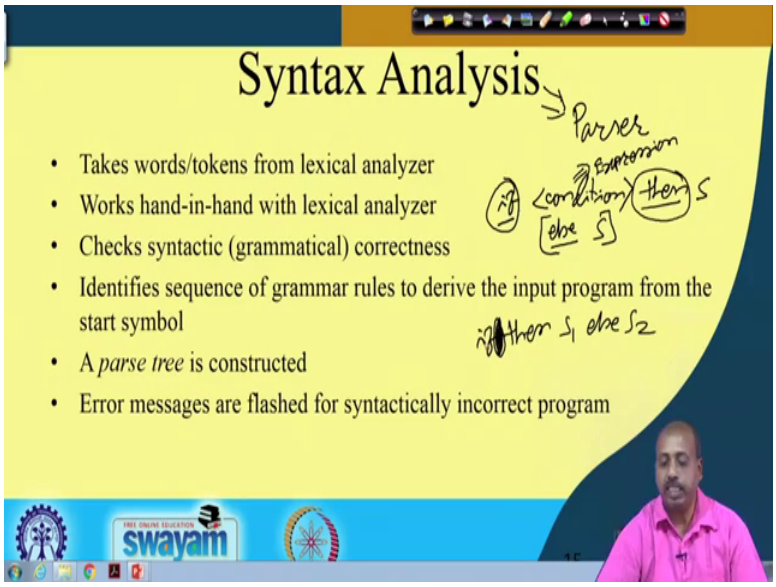
So, from the lexical analysis point onwards so, we can say my program is nothing but a sequence of integers so that is called a token. So, it generate a sequence of integers called tokens to be passed to the syntax analysis phase and naturally later phases did not worry about program text.

So, handling working with characters is a problem because there may be spaces and all. So, once it is a number it is sequence of numbers the handling then becomes easier. So, this is we will see that in detail those that is all the responsibility of lexical analysis phase.

So, it is generally implemented as a finite automata, so as I said that compiler subject is very much dependent on the automata theory. And this lexical analysis tool, it uses the finite automata for design for it is construction.

And we will see that there are different types of finite automata non deterministic finite automata, deterministic finite automata and also this lexical analysis tool can be built around those finite automata.

(Refer Slide Time: 13:00)



So, after the lexical analysis phase the next stage is syntax analysis. So, the syntax analysis phase it is also known as parser. So, this phase is also known as parser. So, we will be using these 2 terms interchangeably just remember that they mean the same thing.

So, this parser it takes words or tokens from lexical analyzer, and they work hand in hand, so it is not that this first the lexical analyzer will generate all the tokens that is there in the program; then the parser will start working on that it is not like that. So, the way it operates is the parser starts as and when it needs to know what is the next token.

So, it will ask the lexical analysis tool what is the next token, so accordingly that lexical analyzer will scan the input, and it will return the next probable token. So, that way it goes, so they work hand in hand with each other this lexical analysis and syntax analysis.

The major responsibility of the syntax analysis phase is to check for syntactic correctness, grammatical correction. So, for example, if you look into any language so, it will say that for example, I have got an if then else statement so it is said that first this q r if should appear, then some condition should come, then some condition should come,

then the keyword then should come, then I can have some statements else some statements. So, out of that this else statements of this part is optional ok.

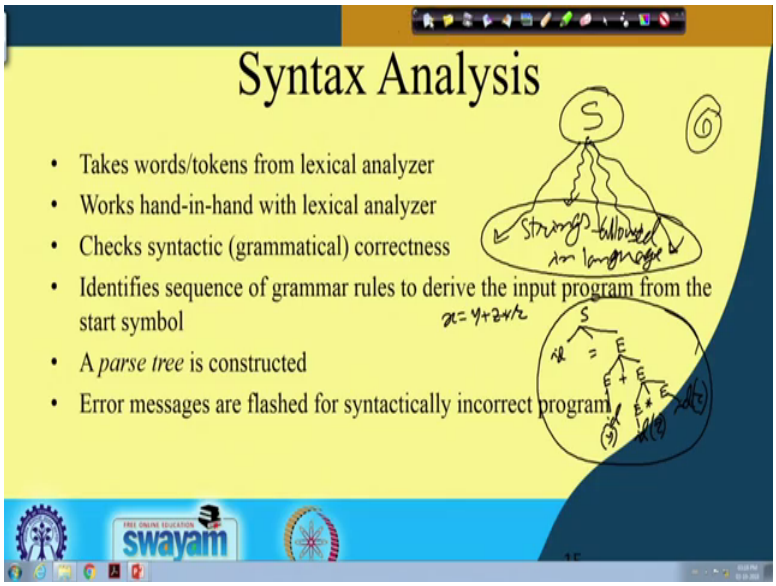
Now, if see this if then else status so this is a grammar so it starts with the keyword if then this condition. So, condition is nothing but some expression and what type of expression is supported in the language so that is also depicted by the language designer. So, they have told what can be the expression, so the a conditional expression so that will be there. Then this keyword then should appear then we can then we should have some statement.

And after that we have an optional else parts so if this optional part is present then this else q r must be present. So, that is the grammar rule so we do not have any grammar rule I cannot write like this if then S 1 else S 2. So, this cannot be written because grammatical this is correct incorrect because in between I needed a condition so that is missing.

So, the syntax analysis phase will actually identify this type of mistakes if the whether the program is syntactically correct or not or grammatically correct or not and if the program is grammatically correct if the input is grammatically correct then it will identify a sequence of grammar rules to derived the input program from the start symbol.

So, as I said that the language is specified by a grammar and every grammar has got a start symbol. So, that all strings that are allowed in that the possible in that language they can be derived from that start symbol using the grammar rules.

(Refer Slide Time: 16:21)



So, there is a special start symbol say S. So, if this is the set of all strings that are allowed in the language. So, these are the strings allowed in the language in the language L. Then, if you use the grammar rules so if G is the set of grammar rules so you can use this set G so that you can have derivation to all of them from this start symbol and in the derivation process it uses the grammar rules of G.

So, then this S is called the start symbol of the grammar. So, all valid programs they can be derived from the start symbol, on the other hand if the program is syntactically incorrect then you cannot derive the program from the start symbol of the grammar.

So, that is the idea so theoretical, the automata theory so it will allow us to do this particular exercise. It will give us a tool by which if a program is syntactically correct. So, we will be able to derive the full text of the program starting with the start symbol of the grammar. On the other hand, if the program is syntactically incorrect it will not be possible to derive the program text starting with the start symbol of the grammar.

So, that is the point or that is the advantage of this syntax analysis phase. And if the problem is syntactically correct so it will construct a parse tree. So, parse tree is basically telling us how this program can be derived for example, if suppose I have got an expression x equal to y plus z into r then you can say that as if this can be derived like this, that I have got a statement for an identifier equal to sum expression.

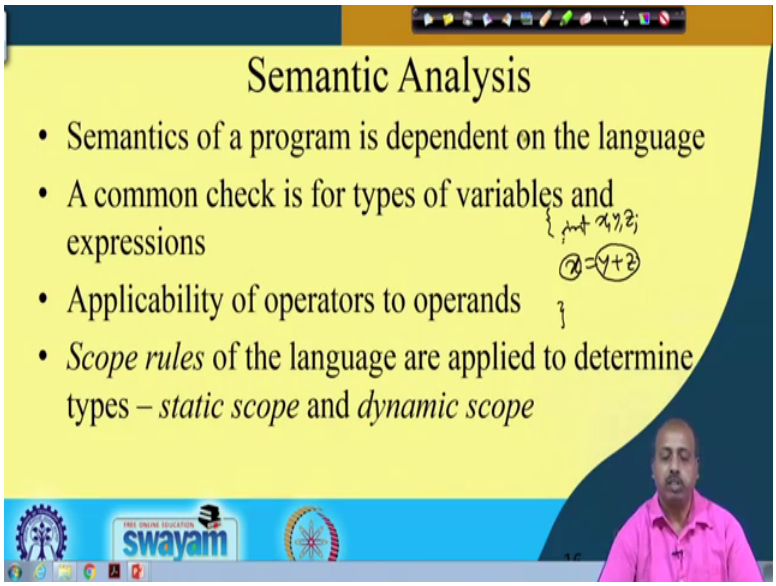
Where this expression is expression plus expression, this expression is giving me an id which is y and this expression is giving me another expression multiplied by expression and this expression is id which is z, and this expression is another id which is r ok.

So, this way we will see that this type of trees can be drawn, where starting with the start symbol of the grammar will be able to derive the whole string. So, this is called the parse tree so we will come to this again later when we go to the syntax analysis phase.

And in this apart from this generation of parse tree so, for the correct programs it will generate the parse tree, so what about incorrect program? For incorrect programs it will flash error messages that so that the user can rectify those errors and then you can again give the job for compilation.

So, this error message design error message display pointing the actual errors and identifying more number of errors in a single pass, so they at the challenges that we have in the syntax analysis phase. So, next comes the semantic analysis.

(Refer Slide Time: 19:50)



So, semantics of a program is dependent on the language, so for the same sentence or similar sentence to languages name in 2 different functions ok. So every program is nothing but some computational some function. So, whatever input you are taking the program is transforming the input to some output so in some sense the program is a function. So, that function is specified by means of the program statements.

So, that meaning, that function that we are talking about. So, that is the semantics of the program. So the meaning of the program, so meaning of the program is the function that it is executing that it is representing.

So, semantics of a program is dependent on the language, a common check is for types of variables and expressions. So, this is a very common sort of thing that we have you know almost all the programming languages where type check is mandatory. Why? Because so for certain operations you cannot do on a certain type of variable; for example, you have got this integer division and integer reminder operation so which are not applicable for real numbers.

On string variables, you can have string concatenation you can have search for a substring and all, which are not applicable if you are taking an integer variable or a real variable like that.

So, if the user of the programmer has done some mistake and has got has written some something which is wrong from the point of view of types of the variables and expressions so that should be caught. So, a common check is of the semantic analysis phase is for type checking, where we essentially check the applicability of operators to operands.

There are certain scope rules of language that are applied to determined the types, like say for example, at some points suppose I write say x equal to y plus z fine. Now it is desirable that these expression y plus z it is type should match with the type of x, but how do we know what is the type of y, what is the type of z, what is the type of x.

So, you say that if this whole thing is within a function then somewhere earlier so, the x x, y and z have been they have been declared. For example, there may be declaration like integer xyz ok. So, you can take that so you get an idea like what is the type of x, y and z. So, when these types of this variables can be obtained at the time of compilation itself. So, that is called static scope, the scope of the variable or a scope of a definition it is static.

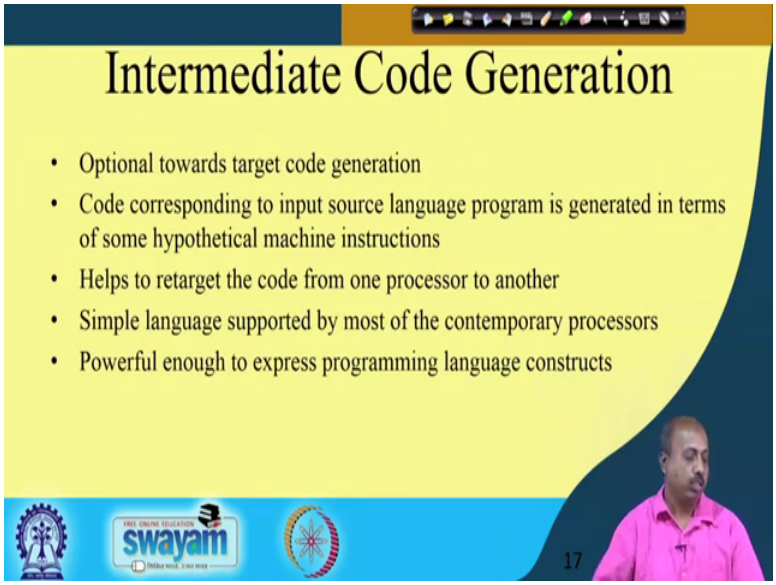
So within this function the x y z all of type integer; however, there is there is another type of scope rules which says that it depends on the execution sequence in which the functions are invoke so, that will determine the values of x, y and z.

For example, in c language you know that if I do not declare the type of a variable in a function and if the variable is as a available globally, then the compiler will take the type of the variable as whatever is defined globally. However, some programming languages that allows nesting of this functions or nesting of procedures.

So, if something is defined some variable is used somewhere whose type is not available within the function, then it will go up and go up in the hierarchy and it will try to identify the block at which this particular variable has been defined. So, that way this nesting of this programming language structures. So, that will determine the actual value actual type of the variable that we are going to use.

So, that actually comes under the scope rules, so, we will discusses about these in detail when we go to the type checking and all. So, for the time being so you know that programming languages they define some scope rules in their definition in their specification and as a compiler designer we have to follow those scope rules for code generation.

(Refer Slide Time: 24:22)



So next we will be looking into so till this much so if a program is semantically correct; that means. So, we there is no error which is done by the programmer at this point. So, it can go into the code generation phase.

So, this code generation phase it as I say that it goes through 2 different stages; one is called intermediate code generation, another is called the target code generation. So, in the intermediary code generation so we use some hypothetical language and code is generated in terms of that language. So, why is it done? So we will see it very shortly.

So, this is and as such this intermediate code generation is an optional states, so it is not necessary that you should always generate an intermediate code and then go to target code. So, it is an optional stage and the code that is generated it corresponds to input source language is generated in terms of some hypothetical machine instructions and that is up to the compiler designer to assume the language, the statements of that language. And we will see some types of this intermediary language later in our course.

So, you will see that we have got flexibility like we can say that my language will support this so type of constructs, and accordingly the code will be generated using that language. The point that helps it that will help us by having this intermediary code generation is that; it will help to retarget the code from one processor to another. So, it is like this, suppose I have designed a compiler which is targeted to say Intel x 86 architecture.

So, for that we know that x 86 has got an instruction set so we can generate code targeting that x 86 processor and the code for that. Tomorrow the same compile if I want to generate code for some say dec alpha machine so that will not possible because the same code will not run there so I have to again do the entire code generation phase.

And that we can save it difficult because now I have to most of the time this code generation is integrated with the parsing or the syntax analysis phase, and then we have to start modifying the syntax analysis phase the from that point onwards so that makes a difficult.

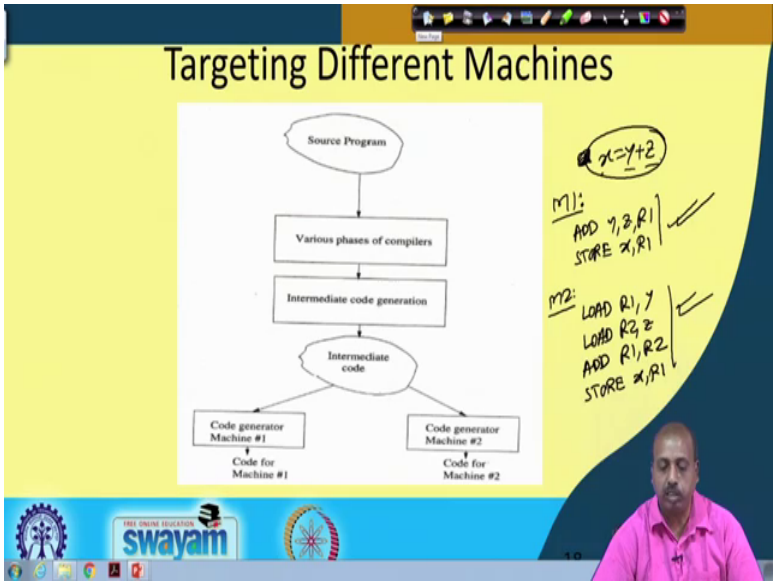
So, what is generally done; this intermediary code generation is that so which once you have a code in that intermediary format so from there it is just a template substitution sort of thing. So, for each intermediary language statement, you can have a template in terms of the target language code, targets language code and then you can just do this template substitution to get the target language program.

So, the retargeting the compiler becomes very easy, from one processor to another processor. Now what about the power of this intermediary language? So, it must be power full enough to express programming language constant.

So, we will that if it is far away it is very very if it is met to simple then it will not be able to catch the programming language constructs or it will not be able to represent the programming language constructs very easily so it will be difficult there. At the same time if it is at a very high level then also there is problem because it will be far away from the machine language code; that machine language of the processor. And then it is another compilers job to translate the intermediary code to the machine language code. So, both way we have got difficulty.

So, you have to do something so that we are add some intermediary stage which is equidistant you can say it is more or less equidistant from this source code and the machine code, so we will see them slowly.

(Refer Slide Time: 28:21)



So, this is the thing that I was talking about. So, we have got a source program that is source program passes through various phases of compilers. And after that it comes to intermediary code generation. So, this intermediate code generation phase produces an intermediate code.

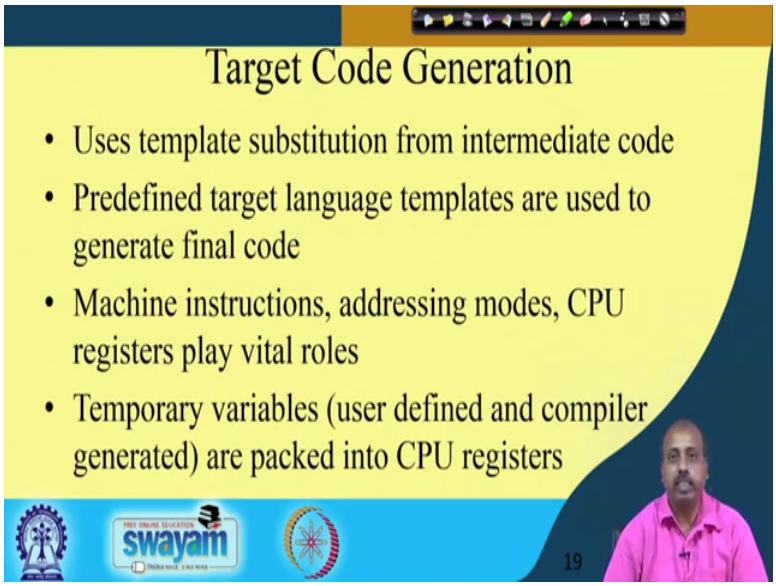
So, this source language program after going through this compilation phase so it has come to an intermediate code representation. Now, from this point onwards so if I am trying for trying to generate code for machine 1.

So I can have a small piece of code which will translate this intermediary code into this code for the machine 1. If I want to retarget the code to target to machine 2. So, that can also be done by developing a small translated here as I was telling that this is simply a template substitution sort of thing. So for example, if I have got a statement like say x equal to say, say x equal to y plus z. So, this may be a statement in the intermediary language.

Now, when I am talking about machine 1so machine 1 may be it is a machine where are this y and z can be kept in the memory and the x has to be a register. So, we can say that Add y z R1 and then store x, R 1 where as in M 2 it may so happen that it does not allow this memory operands directly so all the arithmetic operation so they are to be done on registers. So, in that case the code generation will be something like this that LOAD R1 comma y LOAD R 2 comma z ADD R 1, R 2 and then STORE x comma R 1. So, you see that if this is the program, if this is the intermediary language statement that we have.

So, if I have this type of small small templates for addition so this is the template in M 1 this is the template for M2. So, I can do a simple template substitution so this x equal to y plus z is substituted by these template for M 1 and this template for M 2, so you can generate the corresponding code easily. So, I do not need to regenerate the, redo the entire phase of the compiler I do not have to start designing the compiler from the scratch. So, that is the advantage that we have with this intermediary code.

(Refer Slide Time: 31:09)



Now, to coming to the target code generation phase so target code generation is phase is nothing but template substitution from the intermediate code. Predefined target language templates will be used for generating the code, machine instructions addressing mode, CPU registers. So, they will play vital roles. So, here come the architectural input like the architecture designer have told ok.

This is these are the machine instruction, these are the addressing modes, these are the CPU registers which register can be used in which for which purpose and all. So, everything has to be documented and the compiler designer has to know that by heart and then accordingly do the code generation phase, so that is that makes it very critical.

Temporary variables are and user defined and compiler. So, they are generally packed into CPU registers so that this operation can be made faster. So, this target code generation is nothing but some template substitution and these templates will be design very carefully by taking consideration of this machine details like machine instructions addressing modes CPU registers etcetera.