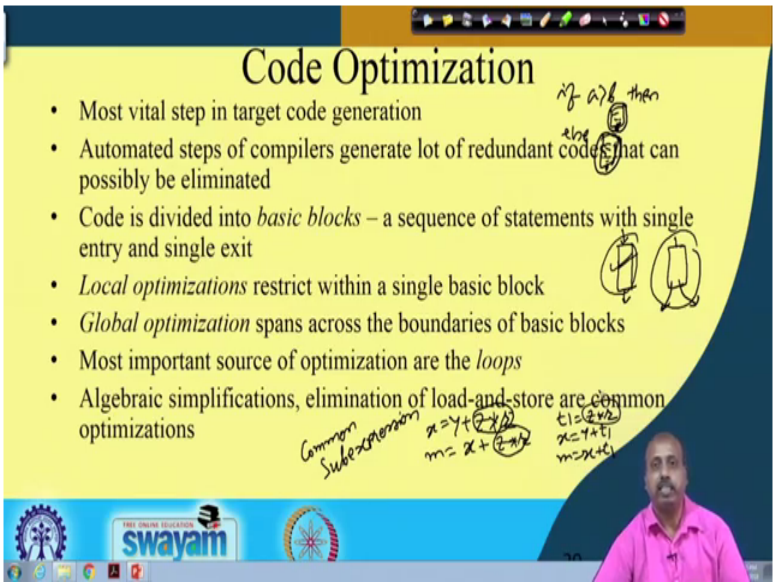
Compiler Design
Prof. Santanu Chattopadhyay
Department of E and EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 04
Introduction (Contd.)

Once the code generation is done, I some sense the job of the compiler is over because, we started with the source code and accordingly we have generated the target code. So, that should be end of the compilation process. But, it really not so and there are many vital operations that goes on after this stage. So, that is known as the code optimization stage.

(Refer Slide Time: 00:40)



Code Optimization

- Most vital step in target code generation
- Automated steps of compilers generate lot of redundant codes that can possibly be eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and single exit
- *Local optimizations* restrict within a single basic block
- *Global optimization* spans across the boundaries of basic blocks
- Most important source of optimization are the *loops*
- Algebraic simplifications, elimination of load-and-store are common optimizations

Now, why this is very vital? This is vital more so, because this whole compilation process is automated. And like if I am given a piece of source language program and I am given the opportunity do hand coding of each instruction in to machine language statements then possibly I can do much better work because we can visualize like forty where is the scope of optimization.

But, when I am doing template substitution sort of translation then there is very much possible, it is very much possible that some operation that we have done in the previous template are to be undone in the next template to do the next job done. So, for example, suppose I have got, I have stored the value of y plus z in to a memory location x in the very next instruction I have to load that value x in to another CPU register.

So, instead of doing it like that maybe I can just store, I can keep a copy of that x in to the CPU register. So, that in the next instruction I do not need to copy it again from the memory, get it again from the memory. So, that way so we have to some sort of optimization. So, this most is a most vital step in target code optimization.

And if you look in to the type of research work that is going on in the domain of compilers so before these optimization remaining part of the process so they are fully automated so there not much scope exists for research on that. But, this code optimization there are many, there are lots of scope for improvement and lot of scope for doing further research and all. So, most of the research works are centered around this code optimization phase but to come to the code optimization phase we have to do the previous stages starting with lexical analysis up to the target code generation. So, that part has to be done, anyway.

So, this automated steps of compilers they generate lot of reddened codes that are that can possibly be eliminated. So, this will be more clear when we go in to the intermediate code generation. We will see that you generate lots of temporaries and those temporaries are just used in the next statement and possibly that temporary could have been eliminated ok. So, we can merge many of those temporaries together.

So, there are lot of redundancy in the code. So, many at many time what happens in that from one intermediately code that is generated it has got a branch statement which makes go to some level l 1 and coming to l 1 we find that there is another go to go to l 2. So, in the first place itself, so we could have made it as go to l 2 so that this is the another extra jump can be eliminated. So, these are all sources of code optimization that can be done. But the code optimization is very difficult in the sense that it takes time. So, if you do not optimize a program then you will see that the compiler runs much faster compare to the case where we have got we ask the compiler to do the optimization.

And often compilers they have got different levels of optimization. So, it is not uncommon to have say about 4 levels of optimization and the user can specify up to which level the compiler will do the optimization. Though in this particular course we do not have scope for go in to code optimization part, but you must keep in mind that this is one of the basic job that almost all the modern compilers they will have in it the optimization part.

So, for the purpose of optimization the target code that we have, so or the machine code that we have. So, they are divided into basic blocks. So, it is difficult like if this entire chunk of program is machine code is given to the optimizer and the optimizer is asked to do the optimization so, it becomes very difficult. Unfortunately this code optimization is a is an np heart problem. So, if you give it any code optimizers, so if you give it a big piece of codes it will not be able to do a good work. So, that is done is that we divide this whole program in to basic blocks.

For example, we can say that suppose I have got a statement like say if a greater than b then I have got a block of statements else. So, here I have got a number of statements else another block of statements so, it may be like this. So, if this block of statement and this block of statement, so if you look into them we will see that they are of this types so they have got a single entry point and a single exit point. Whereas, this if-then-else statement, so it has got it is structure is like this. So, it has got a single entry point but it has got two different exit points. So, it can come up from the then branch it can come up from the else branch. So, it can come up from this point or it can come up from this point. So, this particular structure is more complex compared to say this structure.

So, when we have we have got a program or machine code program. So, we analyze the program and identify this type of blocks the basic blocks ok. So, they are called basic blocks and any basic blocks. So, you can say that code is divided into basic blocks and a basic block is nothing but a sequence of statements with single entry and single exit point. And we do local optimization within this within a single basic block. So, this total optimization process it can be divided in two phases, local optimization and global optimization. So, local optimizations are much simpler and global optimizations are more complex.

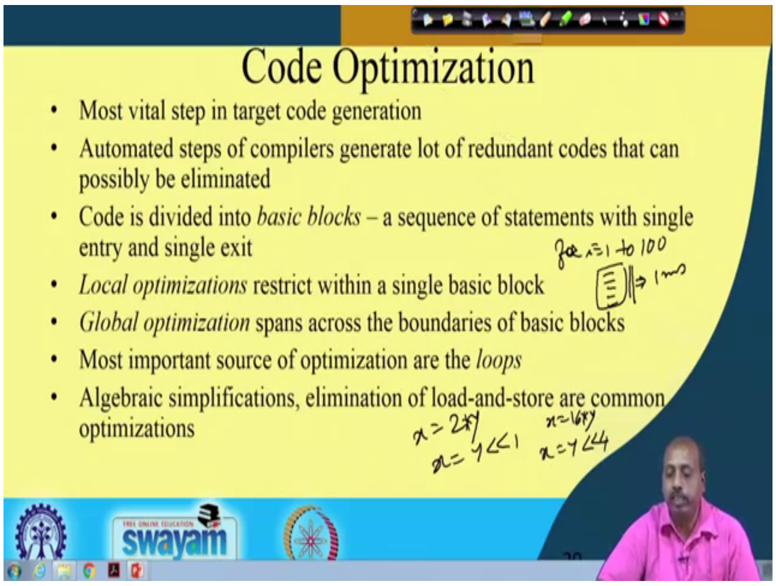
So, first as I said that the compilers may have different levels of optimization at the lower values lower levels of optimization may be it will do only local optimization and as you go to higher levels. So, it will do both local and global optimizations. So, local optimization it will restrict it is view within a single basic block only. So, it will not look across the basic blocks. So, that way if some variable is reused in some other basic block then that is not taken care of in this case.

So, it is all local optimization. So, a typical example of this optimization is suppose I have got two statements like say x equal to y plus z into r and after that we have got another statement m equal to say x plus z into r. Now, between these two statements is z into r. So, this part is common ok so, this is known as common sub expression; so, this is known as common sub expression.

So, a typical optimization that the compilers do is to identify these common sub expressions and evaluate those common sub expressions only one. Like here if I have got a if I take a temporary variable t 1 into which we compute this t 1 equal to z into r, then we write like x equal to y plus t 1 and m equal to x plus t 2 x plus t 1. Then what happens is that this multiplication we do only once. So, we save in terms of the CPU time. So, this is the, this was this common sub expression.

So, when we talk about local optimization we look for common sub expression within the basic block only and when we talk about global optimization we look across the basic blocks. So, across the boundaries of the basic block we try to see whether the variable common sub expressions are existing or not. So, this is just a typical example, there are many other optimizations points at which this local and global optimizations can be tried out.

(Refer Slide Time: 08:53)



Code Optimization

- Most vital step in target code generation
- Automated steps of compilers generate lot of redundant codes that can possibly be eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and single exit
- *Local optimizations* restrict within a single basic block
- *Global optimization* spans across the boundaries of basic blocks
- Most important source of optimization are the *loops*
- Algebraic simplifications, elimination of load-and-store are common optimizations

Now, so, most important source of optimization are the loops, because of the reason that say if I have a loop say I have got a say for loop say for i equal to 1 to 100 I have got a block of statements. Now, you see that if I can if I can optimize this portion ok.

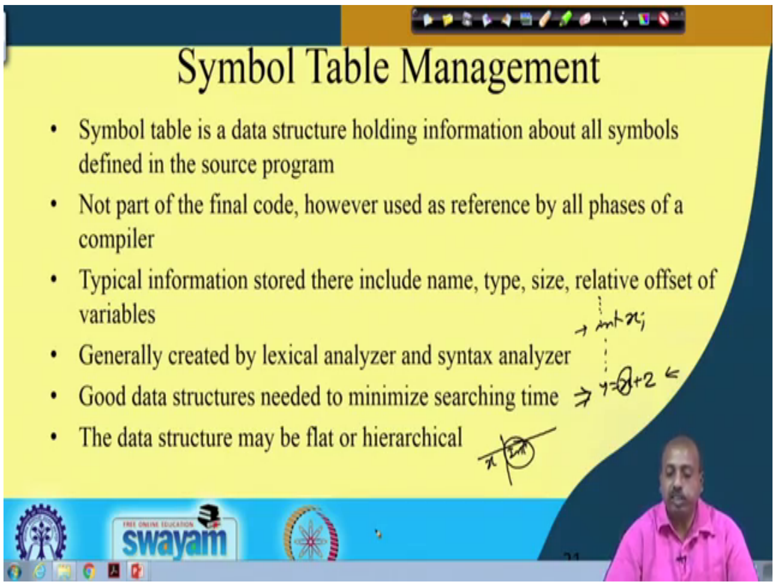
And if I can reduce the execution of this block by say 1 millisecond then since this loop is repeated 100 times, so I have a total saving of 100 milliseconds whereas, if the loop was not there then my saving was only 1 millisecond. So, this way whenever there is a loop and particularly nested loops, so, if you can optimize at the innermost level of the loop then that has got the greatest impact on the overall optimization of the program or overall execution time of the program. So, that is why most important source of optimization are the loops. So, loop optimization is very common and most of the compilers they will do some loop optimization and to come up with the optimized code.

Other, optimizations are like algebraic simplifications like algebraic simplification a typical example is suppose I have got an operation like say x equal to 2 into y. Now, this 2 into y, multiplying y by 2 you know that this multiplication by 2 can be implemented by shifting y by 1 bit, shifting left by 1 bit. Similarly, whenever it is a power of 2 like say x equal to say 16 into y, so you know that I have to shift y by 4 bits to multiply it by 16. So, this way we can, so whenever we have got powers of two for multiplication we can change it to shift operation. So, that is one possibility so, that is one algebraic simplification.

Then elimination of load store; so, sometimes we as I was telling that if in the previous instruction I have computed the value of x and stored in the memory location x and in the very next instruction I am again trying to load the value of x into a CPU register. So, it may be better that we keep the x value of x in the CPU register itself so that I do not need to load it again. So, this load store optimization so, this can also be done.

So, these are various sources of code optimization and there is a huge scope of doing something to the generated code or optimize the generated code so that we get a lot of performance enhancement for the generated code.

(Refer Slide Time: 11:44)



Symbol Table Management

- Symbol table is a data structure holding information about all symbols defined in the source program
- Not part of the final code, however used as reference by all phases of a compiler
- Typical information stored there include name, type, size, relative offset of variables
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical

So, next we will be looking into the module for symbol table management. So, as I said that symbol table management is the symbol table is not directly part of the code that is generated. So, apparently it seems that this is an something extra that we are doing but this helps in the compilation process.

So, by definition symbol table is a data structure that holds information about all symbols defined in the source program. So, it has got all info, all the symbols. So, symbols are like say the variables, constants, procedural names, function names so, they are all symbols or if they go to us, go to as are supported in the program then the levels, so all of them so they are symbols.

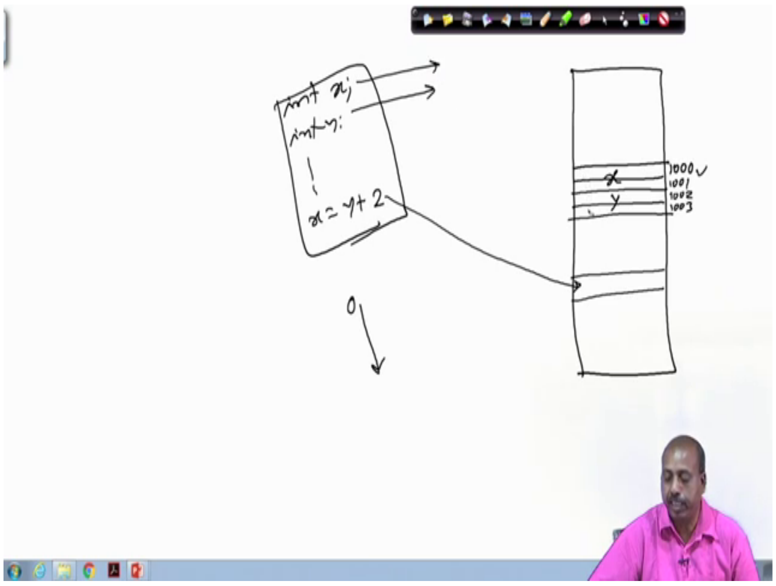
So, these symbols are kept in the symbol table they are noted in the symbol table so that whenever you are doing some later stage like code generation or semantic check things like that, so you can very easily refer to the symbol table and find out the value. So, typically the point is that, maybe while writing the program some time here I have defined integer x. And sometime later I am using like y equal to x plus 2, something like this.

Now, when I am at this point trying to generate code or trying to check the validity of this line. So, I need to know what is the type of x? At that point it is not possible to go back to this line and check what was the type of x. So, instead what is done is that y when this particular definition is seen so, in the symbol table we make an entry for x and note down that it is type is integer. So, that when I am at this line, so I can refer to this symbol table this x is a symbol, so I refer to the symbol table to see the type of it. So, I immediately get that it is type is integer.

So, that is the purpose of symbol table. So, it is not useful no it is not required in the target code final code does not have the symbol table in it, but in the generation process all the references by all phases of a compiler so they will be using the symbol table. Typical information that we store in a symbol table so, they are like this name of the symbol, type of the symbol, size of the symbol relative offset of variable. So, relative offset means from the start of the program at what address or at what value the particular variable comes.

So, it is like this sorry, it is like this. Suppose I have got a variable suppose I have got a program. So, here it starts with the say integer x, then integer y and sometime later I write like x equal to y plus 2.

(Refer Slide Time: 14:34)



Now, you see if this is my program then when the code is generated for this x y etcetera. So, there will be some valid memory locations. So, if this is my memory where this program has been loaded, suppose this program is loaded starting from memory location 1000 ok. And if x is the first variable that we have here so up and assuming that x integer takes 2 bytes, so the bytes 1000 and 1001. So, they are dedicated for x. Similarly, 1002 and 1003 they are dedicated for y; 1002 and 1003 they are dedicated for y.

Now, at some point later, when I have got this x equal to y plus 2 suppose this statement is coded here now whatever code that I put here this day at this point. So, it has to talk in terms of these at this value 1000 of our 1000 and 1002. And, so that because x y does not have any meaning here so I have to talk in terms of this value 1000, 1001, 1002. So, this is true valid if the program is loaded from location 1000.

So, tomorrow if the program is loaded from location 5000 then these values are to be 5000, 5001, 5002 like that. So, what is done? The compiler at the time of compilation it is not possible to know that which address the program will be loaded. So, what we do? We assume that the program is going to be loaded from address 0 and with risk so from this or the 0 is the start of the program. So, from that point onwards where exactly is x defined. So, what is that is called the offset.

So, since x is the very first variable in the program so, a offset of x is 0. If the size of integer is 2, then y is defined at the second address 0 and 1 they are occupied by x. So, 2 will be occupied by y, 2 and 3 so, offset of y is 2. So, this way we can compute the offsets ok. So, this offset computation is done and accordingly we can the code generation stage it can refer to this.

So, symbol table can tell us like what are the offset values. So, this is the, so that is this information is stored in the symbol table. So, relative offset of the variable. Like with the starting with the program at which of which distance from the beginning of the program is the variable stored.

So, it is generally created by the lexical analyzer and syntax analyzer phases and sometimes this syntax analyzer also uses this symbol, symbol table. Lexical analyzer also sometimes uses it, like say suppose whenever it lexical analyzer whenever it finds say another definition of x, so whether the x is already defined or not so that has to be seen.

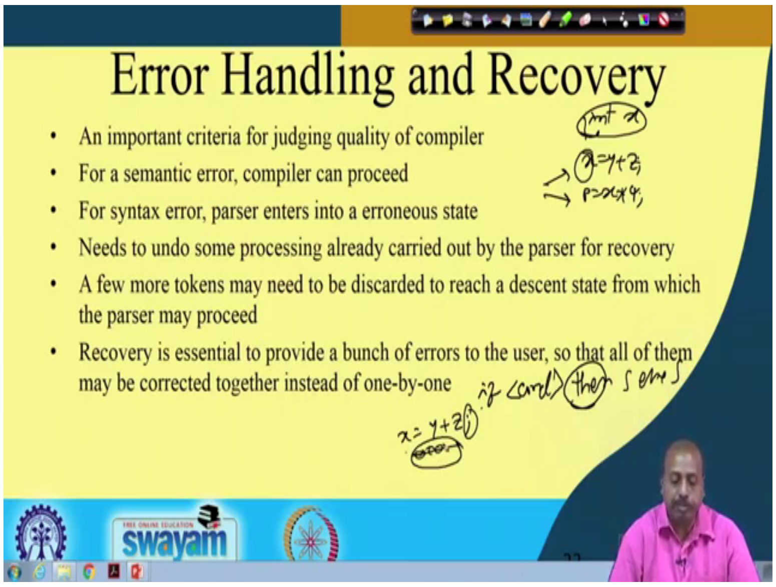
So, duplicate definition checking and also if that may be required. So, then need to refer to symbol table and we need to have good data structure to minimize the searching time because this symbol table search is very common. So, most people but; if as you know that whenever we have got any table the type of operations that we are doing in the table are insertion, deletion and search. So, more generally these are the 3 operations that we do.

And in case of symbol tables, you see that insertion is common but insertion happens and near the beginning of the program code and when you are analyzing the program. So, deletion is very rare because we are not going to delete any symbol from the symbol table possibly and but search is very much. Like whenever I am looking to any program; any program line that we have, so there will be some variables in the program in that line and those variables for those variables we need to identify the type offset etcetera. So, search is very important operation.

So, whenever the way that we represent symbol table or realize symbol table search operation should be made very efficient. So, if we need good data structures that can minimize this searching time whereas, this deletion time we need not be bothered much about it because we do not delete much. Insertion also to some extent it is important so, we need to do that but search is most important.

Data structure that we use for symbol table, so they may be flat or they may be hierarchical in nature. So, when you go to the symbol table management portion so, this will be more clear. So, it will be talking about different types of organizations or symbol table.

(Refer Slide Time: 19:228)



Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

Another very interesting job that the compilers do is the error handling and recovery. So, like how do you judge the quality of a compiler? So, quality of a compiler I mean the target code that is generated what is the quality of the code.

So, quality of the code can be judged by the execution speed of the code, how fast is the code executing, that is more or less depicted by the optimization that the compiler does on the program. Also to some extent it is dependent on the size of the code that is generated. So, how much memory space is needed? Though space may not be a very big criterion now because of all the computers now they have got large amounts for primary and secondary storage but at the initial times so this was a concern ok. So, we can say we have got the memory size as a constraint.

Apart from that, compiler to a good compiler it should help the user in program development. So, for example, when the program is written initially there are lots of bugs in the program, both syntactical bugs and semantic bugs. So, the compiler should be able to catch those bugs very easily and report them to the programmer so that the programmer can correct them.

Now, it is an important criteria for judging the quality of compilers how much error handling it can do how much. So, for example, for first for example I can have you have a situation like this that in my program I am writing like x equal to y plus z, then p equal to x into q like that. But, I have forgotten to define this x, so I assumed; it is for C program it is expected that there x should be declared somewhere r before using them before using it. So, maybe this is missing as a result at this point, this point wherever x appears the compiler will give message that the error message that x is undefined. So, how many such situations the compiler can catch so, that is one problem ok.

Similarly, if I have got an internal statement maybe the growl the language grammar says that it should be like this if condition, then some statement else some statement. May be that I have missed this then the compiler should be able to catch it and then it should give me the information that ok, this is not there, then is expected. So, it should give a hint also. So, it is not that telling that there is an error at line number 30 so that does not help much. So, it should give me an information that then that the compiler was expecting that token then where it called the error.

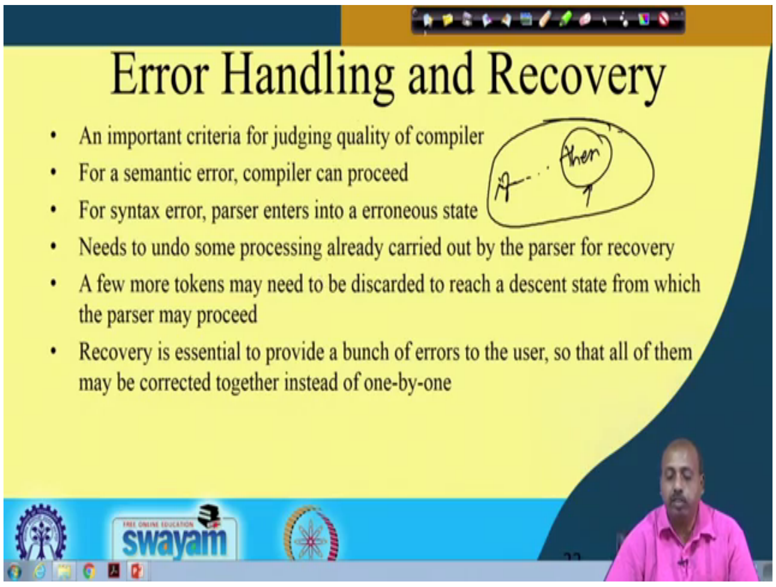
So, then the in that case the programmer can be we can very easily identify the situation, otherwise it is very common to look at the error list that a compiler produces and the programmer says no I have done it why the compiler is giving this error. So, it is definitely there is some problem but it often becomes very much confusing for the programmer from the type of statement type of messages that the compiler generates. So, it becomes confusing for the programmer to identify the actual source of error.

A very common thing is that, some programming languages they if they want that all the statements they should end with a semicolon. Now, if I miss this semicolon and the next statement starts, now if the compiler ideally it should tell me that a semicolon is missing. But, if the compiler cannot pinpoint that situation then what will happen is, it shows some error at the next line because it takes that as if this assignment statement is continuing so it will try to take whatever tokens are coming after this. So, they are also part of the assignment statement. As a result the error message that is given may become confusing.

So, it is not a very easy job, it is not a very easy job to identify or to guess the type of errors that a programmer or a user can do but a good compiler designer should be able to do that guess ok. So, accordingly it should be able to can identify the proper messages that it should flash for to the programmer.

So, that is the thing that the compiler for semantic error, there is a type mismatch and also in that case it does not the program does not violate the grammar of the language. So, in that case it can proceed so, it can process the compiler can proceed but if there is a syntax error then we will see that the parser or the syntax analysis phase itself enters into an erroneous state from which it can it cannot proceed further. So, it needs to undo some processing which is already carried out by the parser for recovery.

(Refer Slide Time: 24:50)



Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

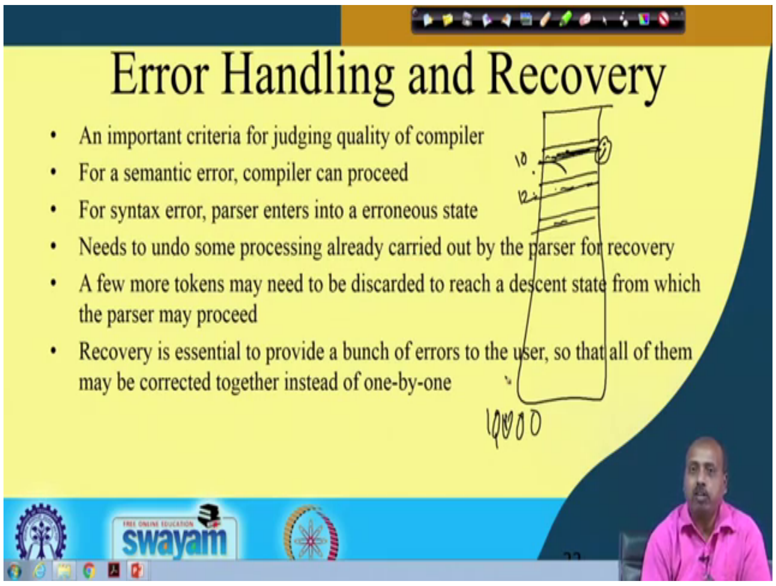
As I was telling that in that if statement that the ‘then’ keyword was missing. Now, what to do ‘then’ with the keyword or ‘then’ keyword is missing. So, we will see later on the say on when you go to the parser design that there are several strategies for handling it. So, one possible strategy is to go back a few tokens and you come to a state as if you have not seen the, if statement at all. So, this entire if statement you skip ok so, that may be one possibility.

So, that the parser which got stuck at this point it can come back to a descent state and from that point it can start again. So, the parser also enters into a erroneous state and in that case it needs to undo some processing already carried out by the parser for recovery.

So, some more for tokens may be needed to be discarded. As I said that if every sentence ends with a semicolon maybe the parser will simply look for a semicolon character until I until and unless it gets a semicolon token it can go on discarding whatever token the lexical analyzer gives it ok. So, that to come to a state where it has seen it to a semicolon token and from that point onwards the person knows that there is a new statement. So, the parser can start accordingly.

So, some tokens may be discarded to reach a decent state from which the parser may proceed. So, this is essential to provide a bunch of errors to the user. Why so much of complexity?

(Refer Slide Time: 26:22)



Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

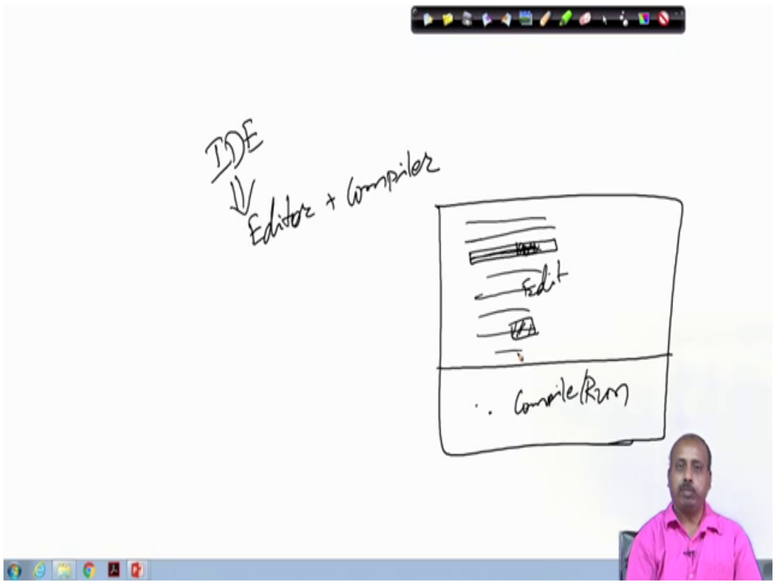
The point is like this, suppose I have I have written a program and this program is very long, so 10000 line code and there is an error at line number 10 ok. So, this is my line number 10, there is an error and when I give the program for compilation it gives me that there is an error in line number 10. I correct that error, again give it for compilation it comes up with a statement that there is an error in line number 12, again after correcting line number 12 it gives me the next error.

So, if it goes like this then you see the amount of effort that the programmer gives immediately the program will tell can the compiler not give me a full list of errors that is there in the problem. So, that I can correct all of them and they are submitted for compilation again. So, why so many times going back editing and then again submitting like that. So, that is very important ok.

So, what happened is that at this point maybe the compiler has come to a state the parser has gone to a state for which from which it needs to do a recovery. So, if it cannot do recovery, so it can stop at this point itself and then go back. Whereas, in case of but to show further errors, so before the user has corrected it. So, somehow the parson needs to recover it needs to recover from this state and do that and then again proceed so that it can catch these other errors also in the program. So, this is very common so, this recovery process is very important, so that it can the parser can come back and come to a decent state from which it can proceeds it.

So, if I say that every statement ends with a semicolon then the this line also ends with the semicolon. So, parser can simply ignore all the tokens steal the semicolon and then start working from the next line onwards. So, that looking into this semicolon it will understand that now the possibly the effect of the error is gone. So, I can start processing the next line. So, that way it can identify the errors that are there at line number 12 or some other future lines. So, this is very common when we have got these compilers which produces a list of errors.

(Refer Slide Time: 28:50)

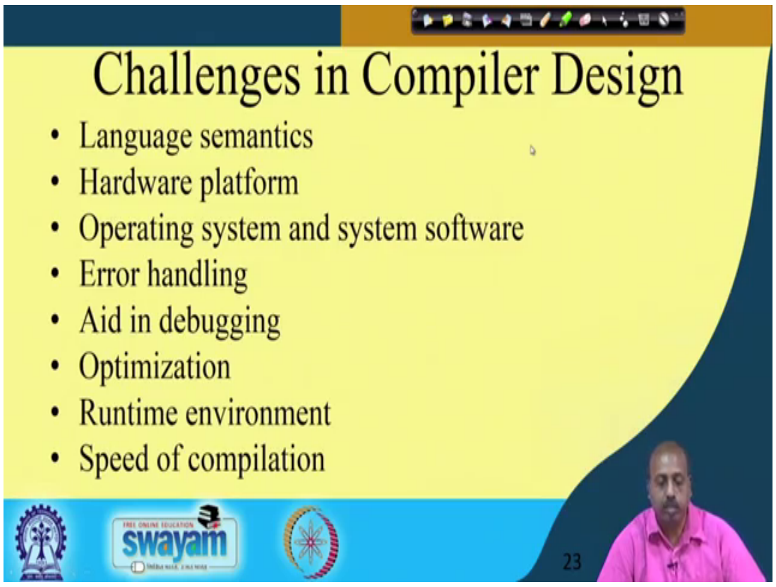


Of course there are compilers which does not give this list which is normally in the situation where we have got this IDE or Integrated Development Environment where this editor and compiler they are together ok. So, this if this is comes up with a screen and in the screen there are portion, one is the editing part this is for the edit this is the edit window and this is the compile window or run window compile run etcetera; many tools have got this type of interface.

So, here I am right I have written the program and then I give it for compilation. And if may say so, maybe the if it identifies an error on this, this line maybe it will just come up with it will come up with this particular line, you highlight this line and maybe a few characters here which is the probable source of error. So, in this case it gives only one error at a time and the user corrects it and then it again goes for compilation and then maybe now it finds that there is an error at this point. So, this is also there so, instead of producing a complete list, it can just show the next error position so, that can also happen.

So, but of course, if I am having a very large program then this is not a very good approach because there are large number of errors may be there. So, the many iterations of corrections will be, has to be they have to be carried out for getting the program syntactically correct. So, recovery is essential for a bunch of errors to the provider to provide a bunch of errors to the user so that all of them may be corrected together instead of one by one. So, this is very important that we should do.

(Refer Slide Time: 30:41)



Challenges in Compiler Design

- Language semantics
- Hardware platform
- Operating system and system software
- Error handling
- Aid in debugging
- Optimization
- Runtime environment
- Speed of compilation

23

Now, what are the challenges that are faced by a compiler designer? Now, apparently it seems that this compiler design. So, it is not that difficult ok. So, if I tell you very frankly, suppose I have got a well designed language then design and I have got a very good idea about the target machine on to which I am going to therefore, which I am going to generate the code it should not take a good a very large amount of time. However, so the problem that we have is that languages they are also very complex they have got many interesting features many interesting functions can be specified there and the target machine also has got many important different type of facilities.

So, if we can exploit those facilities then we can have very good code generated. So, these are the challenges that are faced by the by a compiler designer, the language semantics like the functionality is that you can specify in the language. So, that makes;

the that makes the programmer capable of writing different types of programs with different meanings the compiler designer has to understand has to generate code accordingly.

The hardware platform that we are having, so, there is a hardware platform as we as we vary from one machine to another machine that the architecture itself changes so, naturally we have to do that. Operating system and system software; what is the underlying operating system? So, many of the operations that this programming languages have. For example, some programming language will allow you to do some file handling operation.

Now, the underlying operating system, so that we will also do that will implement this file handling by some methods ok. So, normally they are done by means of system calls and all. So, what are the system calls supported for this file and link? So, that has to be known by the compiler designer and it varies from one ways to another ways. So, the same code for one ways will not work for another one so, we have to be careful there. So, this OS and system software, they will play vital role in the compiler design process then error handling.

So, whether you are going to give a single error or multiple error like that as we have discussed just now. Then aid in debugging, the programs that we have, initial phase of program development they have got errors. So, apart from the syntactical error there are may be logical errors. So, the logical errors if you want to catch then the user will like to execute the program in a single step fashion that is one statement at a time type of execution environment. So, if we want to do that in a compiler has to provide some extra debugging information for that.

In the optimization, so that we have discussed in detail so, type of level of optimization and extent of optimization. Runtime environment; what is the type of procedure called, how are you going to implement the procedural call and also they will determine the runtime environment we will see that later. Speed of compilation; so, how fast is your compiler? So, if it is taking lot of time for compilation maybe if it takes 2-3 minutes then possibly we will say the compiler is very slow and then how to make the compiler fast? So, that is also a challenge.

So, you will see them in successive classes.