Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

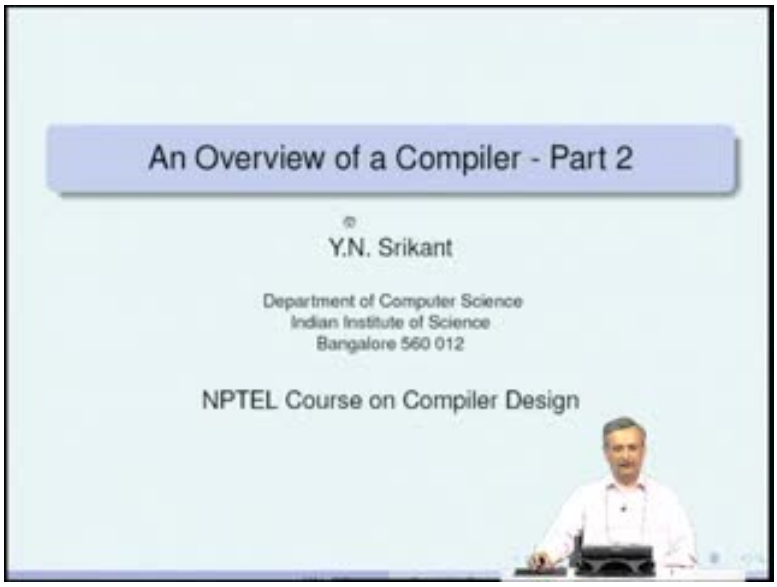
Module No. #01

Lecture No. # 02

An Overview of a Compiler – Part 2 and Run-Time Environments

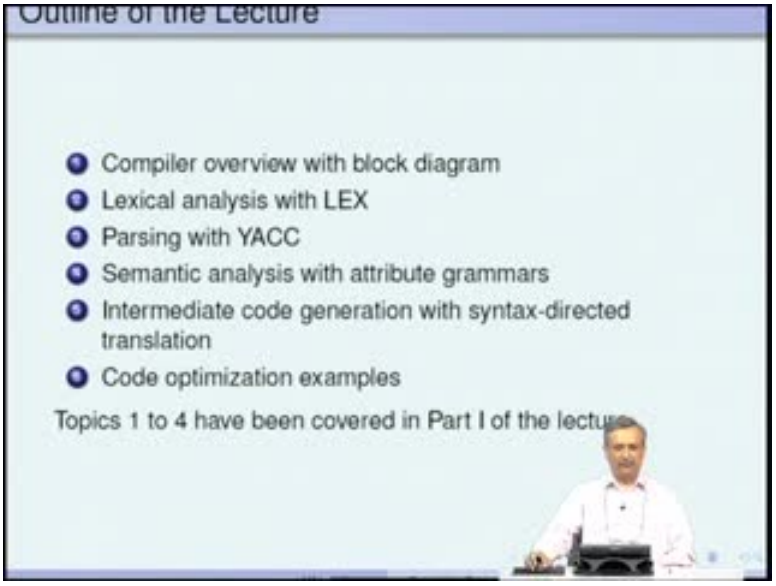
This is part 2 of a compiler overview; we completed part 1 last time. First of all we will have a quick overview of what we did last time and then continue with the second part.

(Refer Slide Time: 00:18)

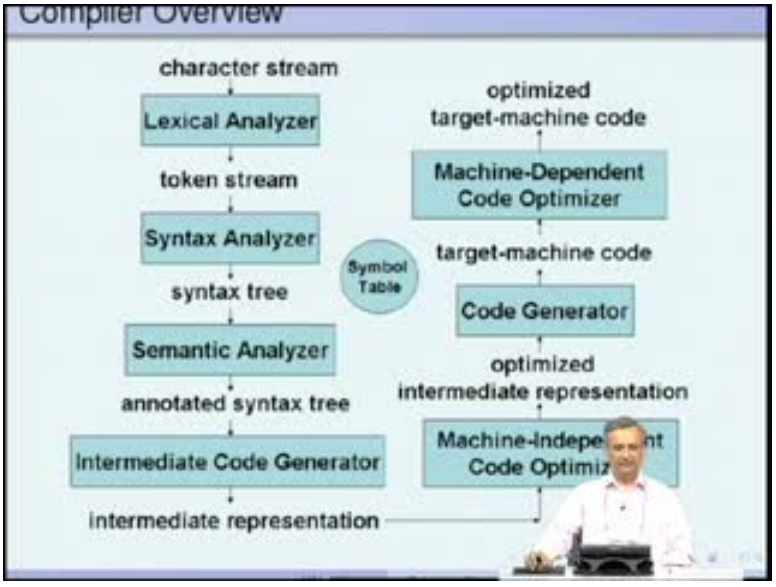


So last time, we looked at the block diagram of a compiler; so this is what we saw, there is a lexical analyzer phase which changes the character stream into a token stream there is a syntax analyzer phase, which changes the set of tokens according to a programming language grammar to a syntax tree and the semantic analyzer changes syntax tree into an annotated syntax tree in which it checks all the semantics of the programming language.

(Refer Slide Time: 00:26)

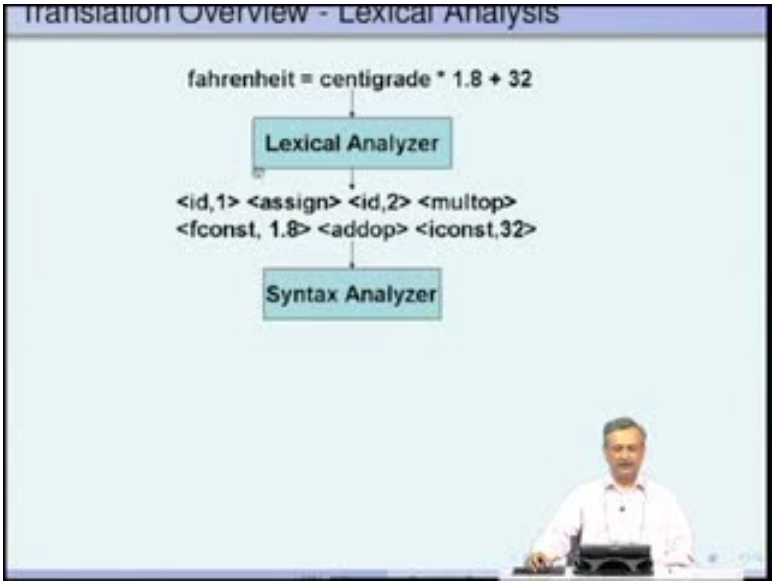


(Refer Slide Time: 00:40)



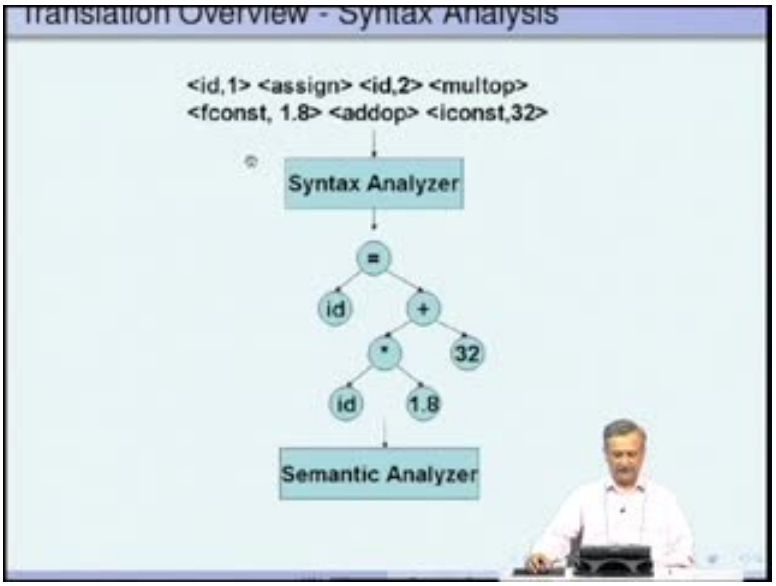
Today, we are going to look at an overview of the intermediate code generator, the machine dependent code optimizer, the code generator, machine independent code optimizer etcetera.

(Refer Slide Time: 01:22)

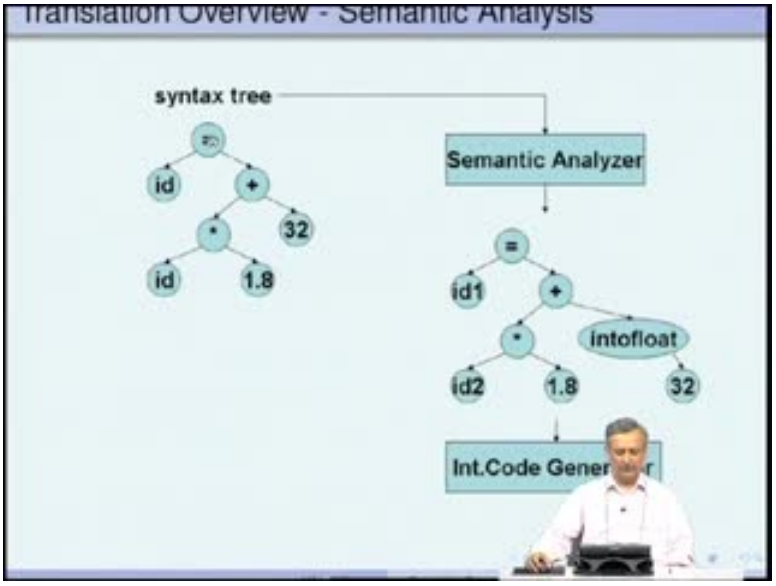


So, this is the slide I showed you last time regarding lexical analysis the input is a an assignment statement Fahrenheit equal to centigrade into 1.8 plus 32 and a sequence of tokens id comma 1 assign id comma 2 multop fconst comma 1.8 addop iconst comma 32 comes out of it.

(Refer Slide Time: 01:47)

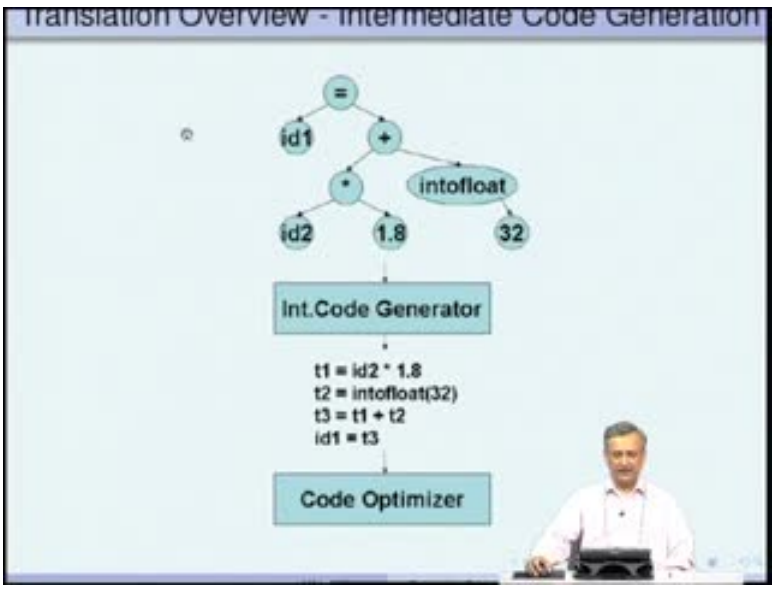


(Refer Slide Time: 02:01)



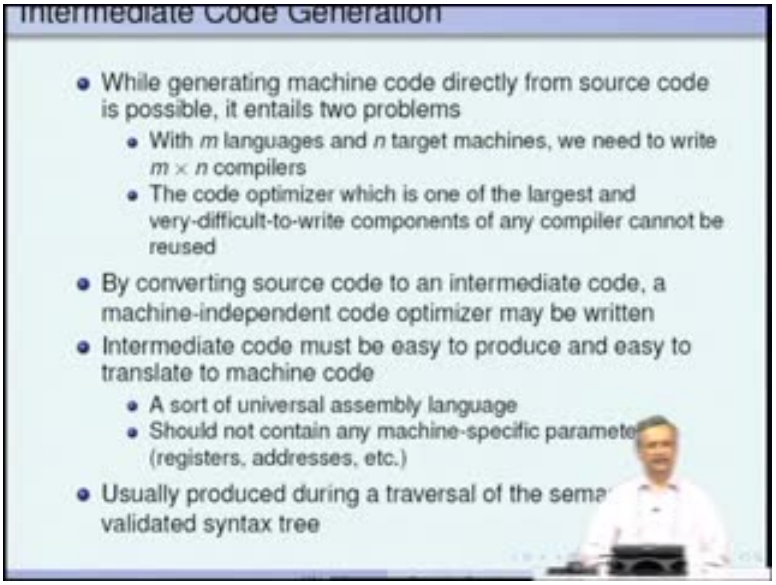
This goes into the syntax analyzer and it change the syntax analyzer parses this particular sequence of tokens and produces the assignment expression tree, as shown here. So the next phase is the semantic analysis phase in which the syntax tree is validated for example, the constant 32 is changed into a floating point constant using the function into float which is inserted into the tree. Similarly, there are checks made whether id and id1 and id2 etcetera are all compatible with the assignment operation and so on. Finally, it goes into the intermediate code generation.

(Refer Slide Time: 02:32)



This is the intermediate code generation overview. So, the semantically validated expression tree or the program tree is the input to the intermediate code generator. The output of the intermediate code generator looks as shown here for example, the bigger expression is broken into smaller expressions, t1 equal to id2 into 1.8 is a small expression; then t2 equal to into float 32 is another assignments statement t3 equal to t1 plus t2 and finally, id1equal to t3.

(Refer Slide Time: 03:14)



So now, let us see, why all these needs to be done in the compiler. While generating machine code directly from the source code is definitely possible, it is not an impossible problem but, when we do so, we lose out many things. There are two problems that are well known for example, if there are m languages and n target machines and we generate compilers directly rather, we write compilers directly which generate machine code from the source language. Then, we need to write m into n compilers, hardly any part of the compiler can be reused and there is a lot of effort in writing a fresh compiler. The first part of a compiler can always be reused that is the lexical analysis, the syntax analysis and the semantic analysis can always be reused, whether we are generating intermediate code or we are generating machine code.

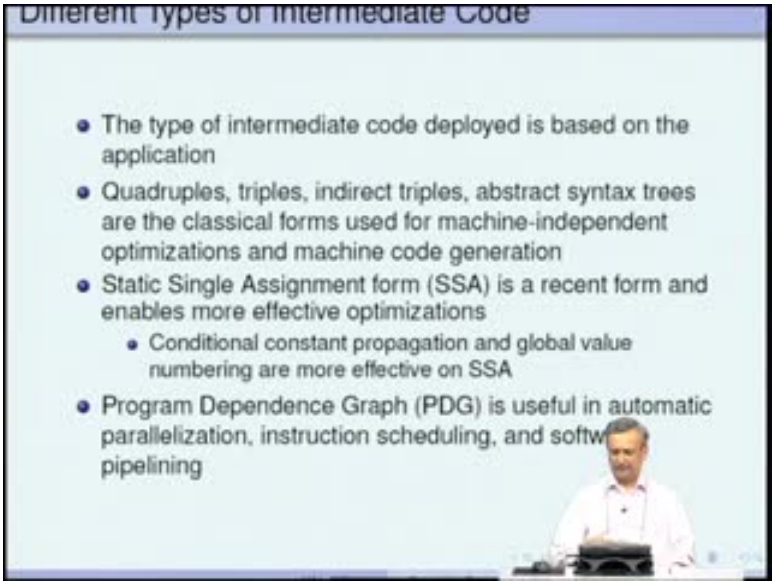
But, if you are generating machine code directly then the code optimizer, which is one of the largest and the most difficult components of any compiler cannot be reused. So what happens is the code optimizer will have to be now targeted towards the machine code

which is being generated and therefore, every machine code has its characteristics which are very different from each other and we will not be able to reuse this particular code optimizer.

Whereas, if we convert source code to an intermediate code, a machine independent code optimizer may be written, so what is this intermediate code? The intermediate code is some sort of a universal assembly language. Of course, it should be easy to produce, it cannot be as difficult as machine code to produce and it must be easy to translate it from intermediate code - any intermediate code to machine code as well.

Another important characteristic is that it should not contain any machine specific parameter such as registers, addresses etcetera. The reason is, if it contains machine specific parameters, the machine independent nature of the code is lost. So we will again be back to square one and we need to write the machine code generator and this even the optimizer etcetera, all over again. So this machine independent code is usually produced during a traversal of the semantically validated tree.

(Refer Slide Time: 06:00)



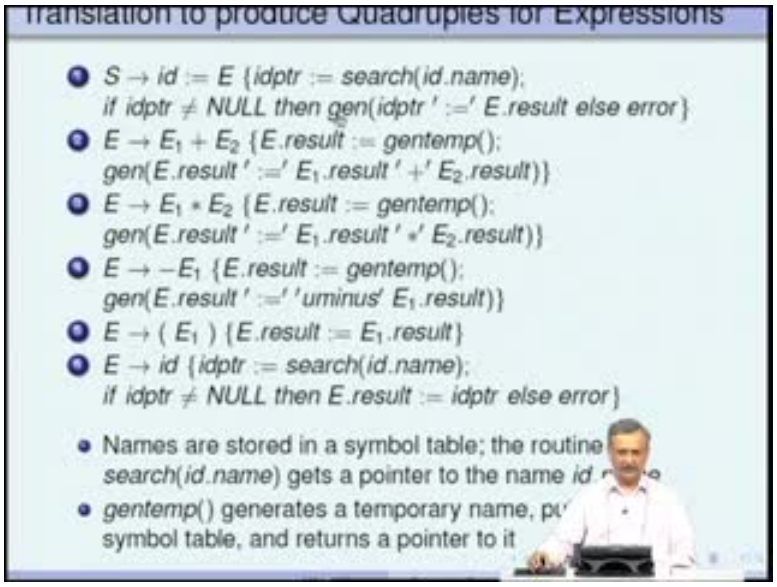
There are different types of intermediate code; it is not that the same intermediate code can be deployed in every application. So, it depends on what type of application we have in mind when we design the intermediate code. For example, quadruples, triples, indirect triples, abstract syntax trees these are some of the classical forms of intermediate code, which are machine independent and then these can be used for machine independent

optimization and also for machine code generation; that is fairly easy, this are very well known and they are used in almost every complier.

There are two other special forms of intermediate code that I am going to mention here. One of them is the Static Single Assignment form called SSA; this is a very recent intermediate form and this is much better than the other type of intermediate code that is used in compilers, because we can do many more optimizations more effectively on this particular static single assignment form. We are going to look at the details of SSA in one of the later lectures. For example, conditional constant propagation and what is known as global value numbering are two optimizations which are far more effective on the SSA form than on any other form.

The other special intermediate code that I mention here is the Program Dependence Graph - PDG as it is called. This is very useful in automatic parallelization instruction scheduling and also software pipelining.

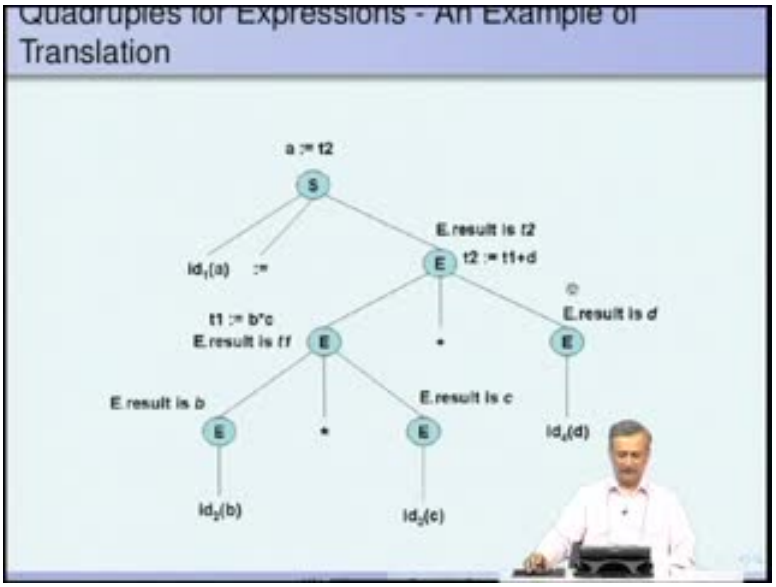
(Refer Slide Time: 07:56)



Here is an example of translation to produce quadruples from arithmetic expressions. So let us go through it step by step. The grammar is a very simple grammar. S going to id equal to E, is the production for assignment. SE going to E_1 plus E_2, is the sum expression. E going to E_1 star E_2, is the multiplication expression. E going to minus $E_1$is the negation; then E going to bracket E_1; bracket is the parenthesize expression. E going to id is the terminating production, which gives you a single variable. So, if this is the

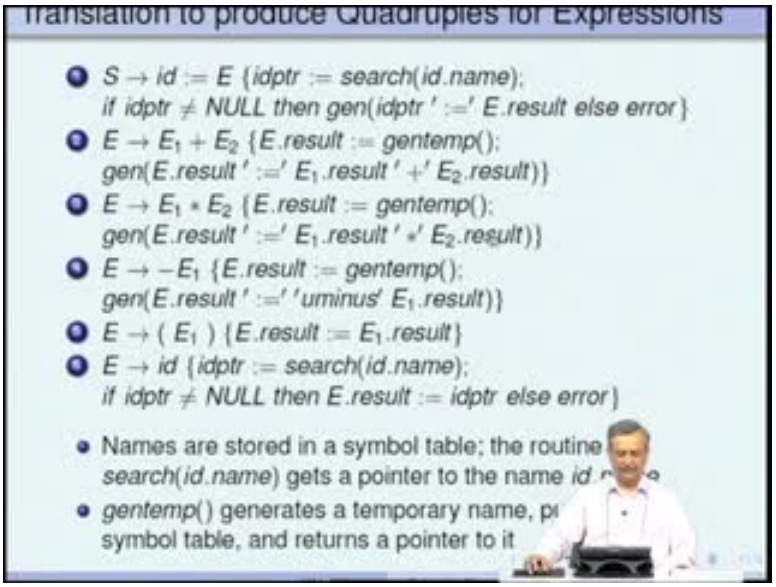
grammar, first of all let us look at the kind of expressions that we produce and then come back to this grammar all over again.

(Refer Slide Time: 08:58)



Here is a nice example, let say we have a simple tree in which a equal to b star c plus d is the expression, so b star c plus d is the expression and it is being assigned to a, so this is our complete assignment statement. Now, let us go back one step and see how this is translated. Before that, please observe the syntax tree that is produced. So at the first level, we have id then equal to an E.

(Refer Slide Time: 10:20)



At the second level, we have E plus E and at the third level, we have E star E and at the fourth level, we have the identifiers. So the code generation the machine the intermediate code generation actually goes in a bottom-up fashion. So for example, the code is generated for E going to E star E first and then, the code intermediate code is generated for E going to E plus E and finally, the intermediate code for S going to id equal to E, so let us see how this happens. So if it is an assignment id equal to E, we search the action indicates that we search the symbol table get the pointer for that particular name from the symbol table and if the name is already present in the symbol table then the pointer is not null; then, we generate just the simple assignment statement in the intermediate code idptr equal to E dot result.

So, E dot result is the place where the result of the arithmetic expression is going to be placed. This E dot result is also a pointer into the symbol table; it is not that it is a machine location. The machine location will come later and it will be assign during the machine code generation phase. So, if the name is not present in the symbol table, obviously it is an error.

Let us take the next production, E going to E_1plus E_2. So in this case, we have the result of E_1 in E_1 dot result; the result of E_2 in E_2 dot result and to add them up, we actually need a temporary. So, the gentemp routine creates a temporary and E dot result equal to gentemp will transfer the pointer into to the temporary variable into the variable E dot result. So gentemp actually generates a temporary name and puts it in the symbol table and also returns a pointer to it that is the way, we have assumed it.

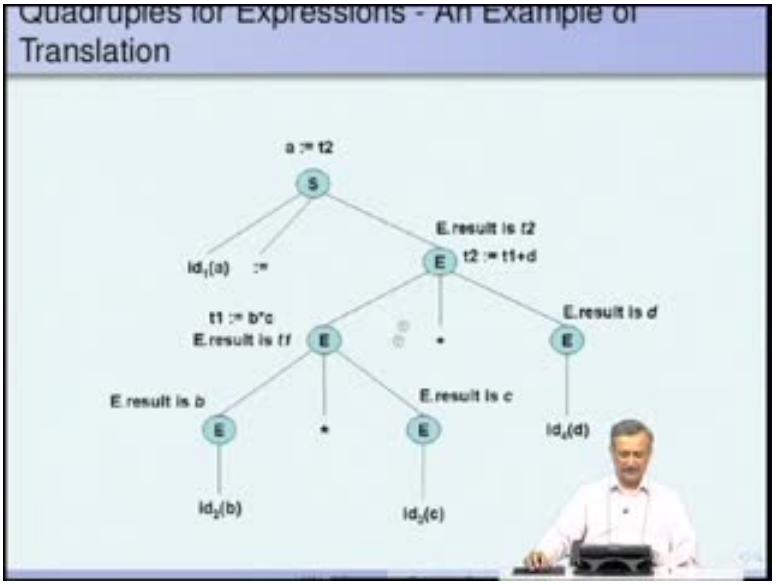
So now, once the temporary is generated; the next is to generate intermediate code E dot result equal to E_1 dot result plus E_2 dot result. So there we have the intermediate code for the arithmetic expression consisting of plus. E one star - E going to E_1 star E_2 is very similar instead of plus there is a star, otherwise there is no difference. Similar is the case of E going to minus E_1 and E going to parenthesis E_1; parenthesis does not require any intermediate code generation it is just that the location, where the result of E_1 is stored is transfer to the location E dot result.

Finally, E going to id again makes us search in the symbol table, gets the pointer to the name id dot name and if it is not null then, the place E dot result is going to be a the pointer to the symbol table, otherwise it is an error. So in other words there is no code

generated for the productions 5 and 6; it is only for the other productions that we have really generated the intermediate code.

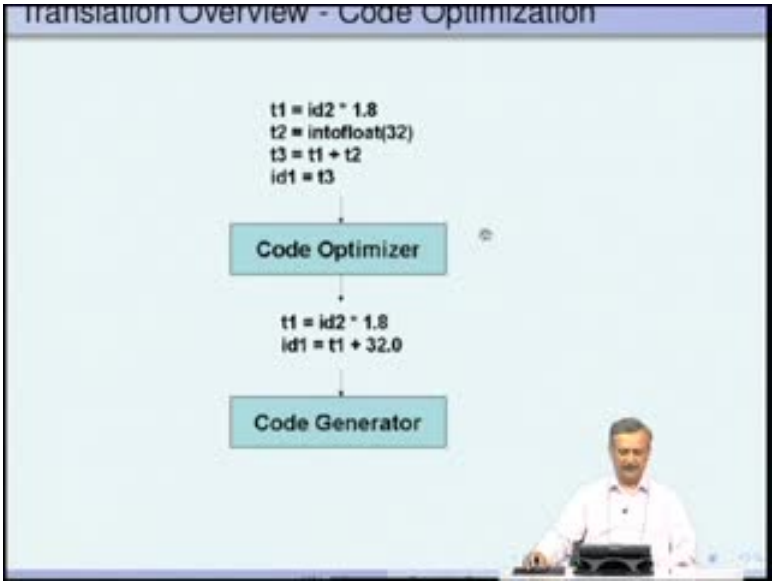
Intermediate code is very similar to the higher level programming language code but, the most important characteristic is that it can have only a single unary operator or a single binary operator; bigger arithmetic expressions have to be broken into smaller ones and finally, temporaries must be used to calculate the result.

(Refer Slide Time: 13:47)



This is very clear from this particular diagram. So, if you look at this expression E going to E star E - E dot result is b for this particular E, E dot result is c for this particular E and here, we generate a new temporary t1 and E dot result will be t1(Refer Slide Time: 14:00). Here, E dot result is d here we generate a new temporary t2 and that will keep the result of t1 plus d and finally, the assignment is a equal to t2. So finally, the intermediate code that we generate would be t1 equal to b star c; then t2 equal to t1 plus d and finally, a equal to t2. These are the three intermediate code statements that would be generated in a bottom-up fashion.

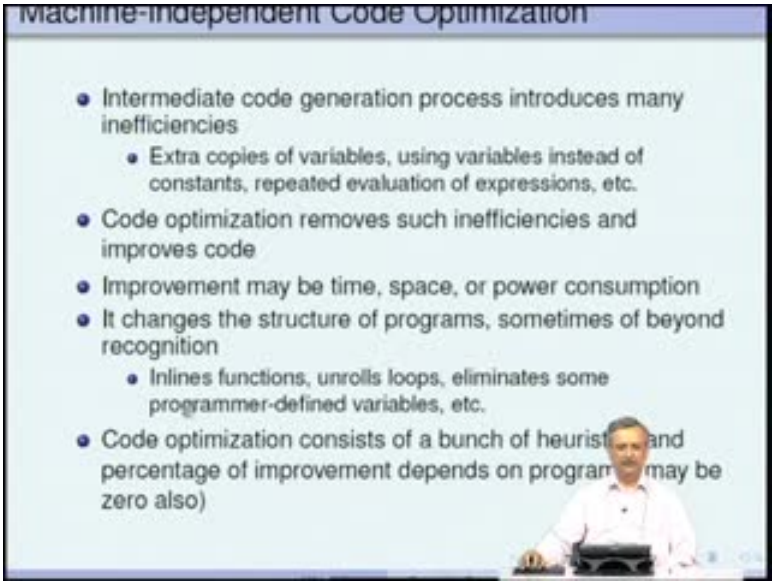
(Refer Slide Time: 14:44)



Now, let us look at the next phase of compilation called the code optimization. This is the input that we got from - rather the output that we generated from the intermediate code generator. Here, we have these statements but, as you can see where is some inefficiency in these. Look at the output here, the output that we generate after the code is optimized; it directly says t1 equal to id2 into 1.8, instead of accumulating id2 into 1.8 in t1 and then rather sorry t2 equal to into float 32 and then t3 equal to t1 plus t2 is directly and then id1 equal to t3 is directly translated into id_1 equal to t1 plus 32.0.

So, the three statements t2 equal to, t3 equal to and id_1 equal to have been combine into a single statement id1 equal to t1 plus 32.0. So, such elimination of unnecessary statements is possible using the code optimizer. Code optimizer improves the quality of the intermediate code that is generated and fits it into the code generator.

(Refer Slide Time: 16:07)



So here is the detail; intermediate code generation process introduces many inefficiencies; this is the reason why we require the code optimizer phase - machine independent code optimizer phase. So extra copies of variables that is, the temporaries, we already saw the t1 t2 etcetera, all temporaries being generated. Then using variables instead of constants, so we would have simply initialize the variable to a constant and use that variable in place of the constant and then there are expressions which are going to be repeatedly evaluated; so these are all the sources of inefficiency, you can eliminate these. If there are extra copies of variables, we can retain one of them and eliminate the rest; instead of variables, we can simply do constant propagation and replace the variable name by the constant value and in the case of repeated evaluation of expressions we evaluate the expression only once and use the same value again and again.

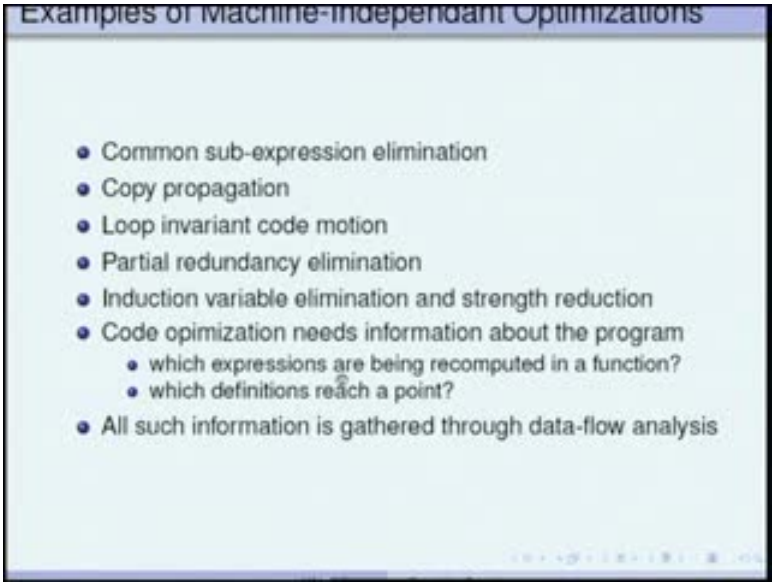
So, code optimization removes such inefficiencies and improves code that is what I just now mentioned. Improvement may be time, space or power consumption for example; we may want to make the code fastest; so in that case, we want to reduce or minimize the amount of time that the code takes. We want to make the code space efficient; for embedded applications the amount of memory that the code takes may be very important and in such case, the number of instructions will have to be reduced in order to save space.

Finally, in the same embedded application such as sensor networks, power consumption

is a very important factor. So, we may want to actually optimize the code to save power instead of time or space. So, it is possible to do optimization in many ways. The code optimizer also changes the structure of the programs; sometimes it is beyond recognition. So for example, the functions are all inline there are no functions any more. The body replaces that function call; unrolls loops. So, if the loop runs from one to ten in one to one million may be I am going to unroll the loop one thousand times and run the loop only from one to one thousand, the rest of it is unrolled and it is going to be a huge body inside the loop and it also eliminates some of the programmer defined variables etcetera. For example, if the same variable the different variables have the same value as I told you, we can use one of them and eliminate the rest.

Code optimization consists of a bunch of heuristics and percentage of improvement depends on the programs. This is a heuristic, so it is not guaranteed to give you any definite result. If the improvement happens then it is very good; if the improvement does not happen even then it is not going to change the functionality of the code that is the best guarantee that I can give you but, on the average most of the programs benefit by code optimization.

(Refer Slide Time: 19:31)



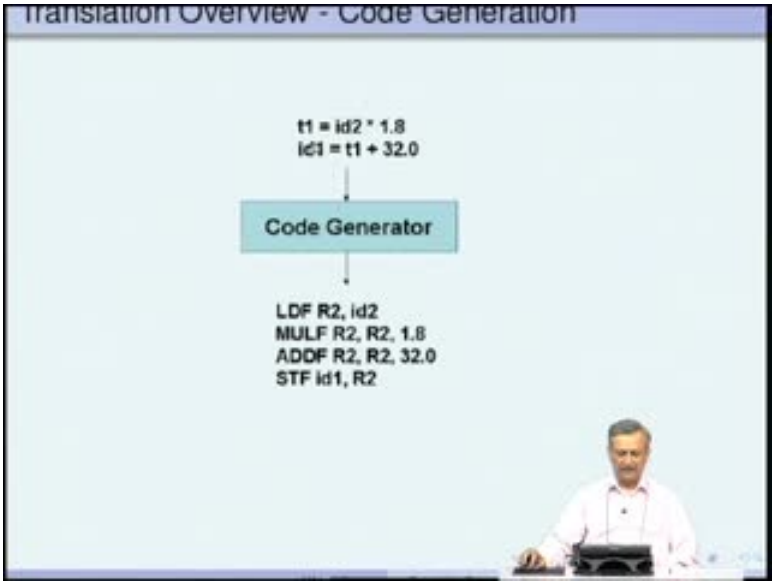
Then, here are some of the examples of machine independent optimizations, some of them I already mentioned for example, common sub expression elimination the same expression need not be evaluated again and again. Copy propagation so I do not have

several variables carry the same value so I do not have to keep all of them I can just use one of them.

Loop invariant code motion: there is a lot of code inside a loop which is not altered inside then in that case, the code can be moved outside the loop. Partial redundancy elimination is a bit complex to explain but, it is one of the most useful optimizations and we will be looking at it in detail in one of the late lectures.

Induction variable elimination and strength reduction are similar; they actually are carried out in a loop. So multiplications can be change to addition and induction variables which are nothing but, incrementing variables sometimes can be eliminated, code optimization needs information about the program, where this is very obvious and the compiler actually obtains this information using a process called data flow analysis, so for example which expressions are recomputed in a function, which definitions reach a point. These are some of the optimizations possible; the information that are necessary for an optimization.

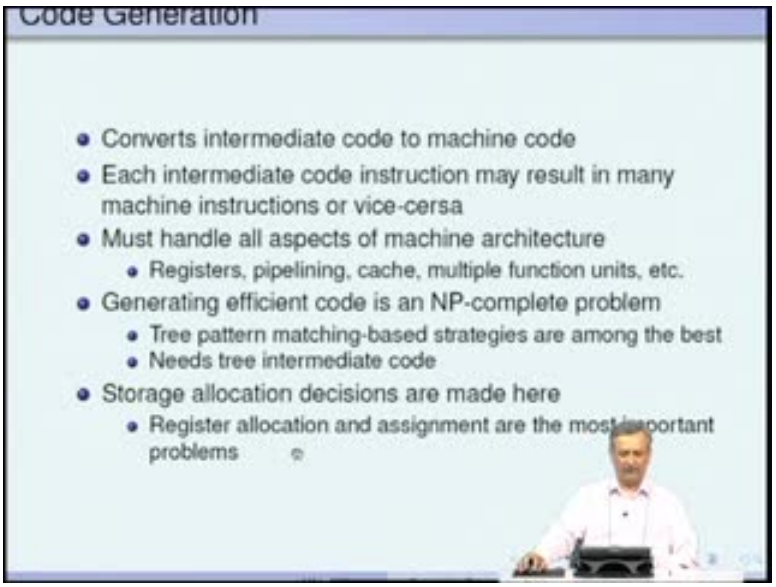
(Refer Slide Time: 21:08)



Next, the code generation process is, it follows the code optimization process. For example, the intermediate code, which is improved in a code optimizer, is input to the machine code generator and here is a sample of the machine code. The assignment statement t1 equal to id2 star 1.8 is translated into load floating point R2 comma id2 and then multiply floating point R2 comma R2 comma 1.8. Now R2 will contain the value of

the expression id_2 star 1.8 and now, instead of id1 equal to t1 plus 32.0. Since the value of t1 is already in the register R2, we can simply say, add floating point R2 comma R2 comma 32.0 and finally, store floating point id1 comma R2 will assign the value of this entire expression to id_1. This is the sample machine code which is generated in compiler.

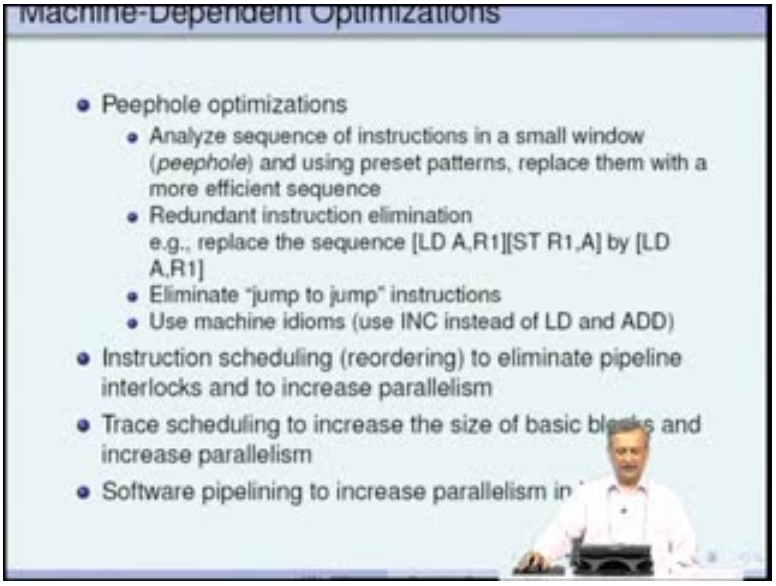
(Refer Slide Time: 22:14)



So here is some detail, a machine code generator converts intermediate code to machine code. Each intermediate code instruction may result in many machine instructions or vice versa, it is possible sometimes to compress many intermediate instructions into one machine instruction as well. It must also handle all aspects of machine architecture for example, we must use the registers very efficiently, try to use as many registers as possible because a registers are very fast; it must take care of the pipelining nature of the processor, then it must handle the cache sometimes; then if there are multiple function units, it must put them to efficient use. It should not be that only one of the function units is used and rest are all idle etcetera.

Generating efficient code is an NP-complete problem, so it is one of those problems which is extra difficult. There are many techniques, which are used to simplify this particular operation for example, tree pattern matching based strategies are the best strategies we look at these in detail later; this needs, what is known as tree intermediate code instead of a quadruples and which we have seen so far.

(Refer Slide Time: 23:52)



Storage allocation decisions are also made here. Register allocation and assignment are the most important problems here, so which resistors use which are the variables that can be placed in these registers are there the best allocations that we can do etcetera are the questions we answered during this particular phase of compilation. Then we also have machine dependent optimizations; we saw machine independent optimizations a few minutes ago and now, we see some of the machine independent optimizations. For example, what are known as peephole optimizations? These peephole optimizations they analyze a sequence of instructions in a small window which is called as a peephole and using preset patterns, replace them with more efficient sequence. So it may be that a particular sequence of instructions is inefficient and there is a way of writing them in a efficient fashion, which is known to all programmers, such patterns can be put into a peephole optimizer.

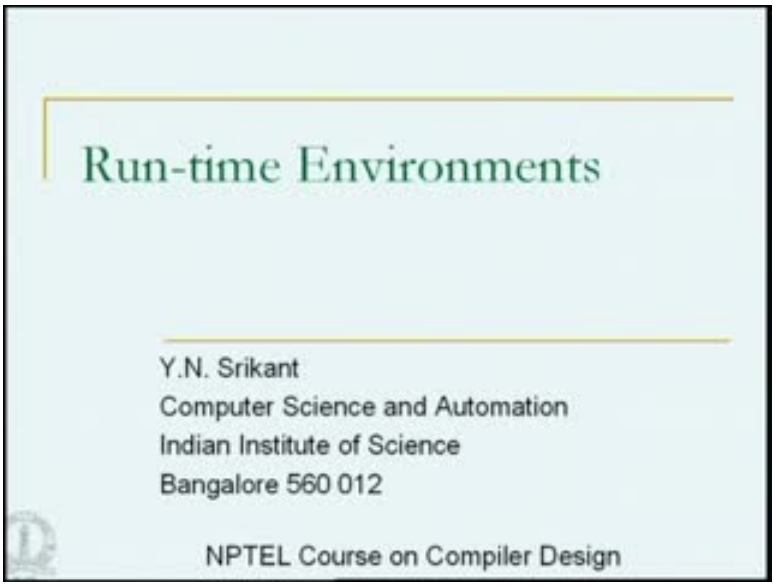
Redundant instruction elimination here is an example, if you have a sequence load A comma R1 followed by store R1 comma A; then actually this is not necessary- it is not necessary to have this store instruction after all because, A already has the value that we require and therefore, these two instructions may be replaced by load A comma R1 itself.

The eliminate 'jump to jump' instructions: So, there is a jump instruction which does not execute any other code but, just jumps to another jump instruction; such multiple jumps can be eliminated, we will see a little bit of this later on. It is also possible to use

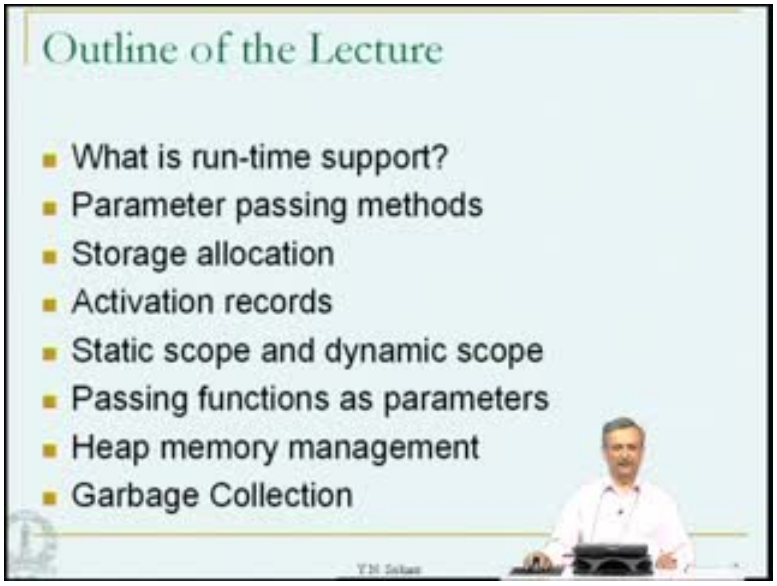
machine instructions- idioms for example, use increment instead of load followed by add etcetera. So instruction scheduling is a very important machine dependent optimization; it is used to eliminate pipeline interlocks and increase the parallelism in programs; it is nothing but reordering instructions in order to reduce the pipeline stalls.

Trace scheduling is used to increase the size of basic blocks and then increase the parallelism. Finally, software pipelining is used to increase parallelism in loops. So these are some of the examples of machine independent optimizations that we have and this brings us to the end of the overview of a compiler.

(Refer Slide Time: 26:15)

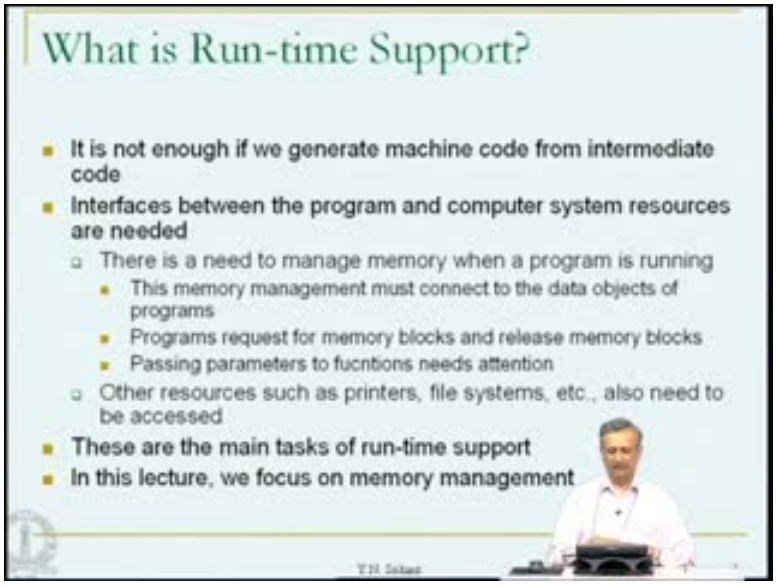


(Refer Slide Time: 26:25)



So, now we begin the next lecture on runtime environments. Here is an outline of the lecture; let us see what runtime support is; then we are going to see a few of the parameter passing methods. So we will discuss storage allocation; what are known as activation records will be used throughout this lecture, so we have to understand what they are. We discuss static scope and dynamic scope; what are these? Why are they necessary? How to handle them, and so on. How to pass functions as parameters to other functions? Then heap management - heap memory management and garbage collection, one of the techniques which is possible to discuss in the limited time.

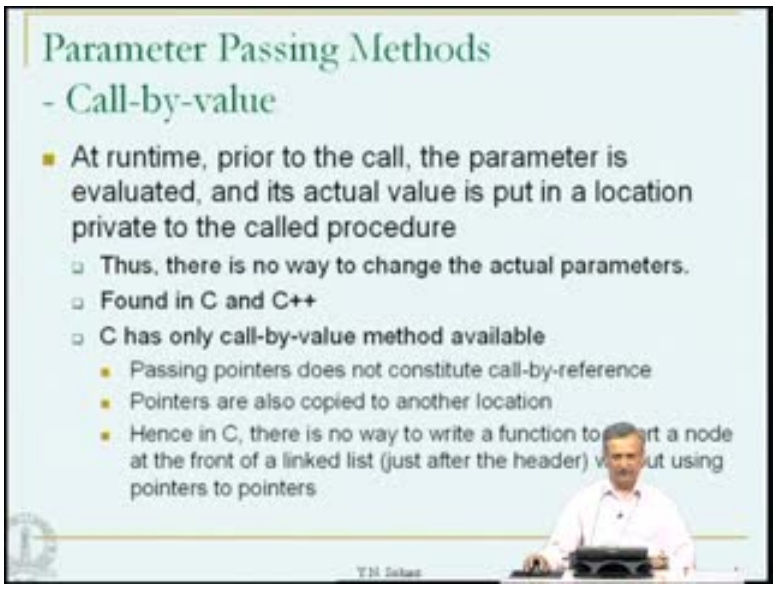
(Refer Slide Time: 27:15)



So what is runtime support? See, it is not enough if we generate machine code from intermediate code and then hope for the best; it does not run at all. Interfaces between the program and the computer system resources are also necessary, otherwise just dumping the code which is produced into memory and then trying to execute, it will never work. There is a need to manage memory, when a program is running. So for example, this memory management must connect to the data objects of the program. So, it must understand, how to fetch values of the variables and then if there are parameters, how to pass them to functions and then where to store the return address, if there is a subroutine jump and how to access the function result after the function returns etcetera.

So, programs normally request for memory blocks and then they go on releasing memory blocks whenever they are of not much use. So, all such memory functions must be taken care of by the memory manager. Other resources such as printers, file systems etcetera are also to be accessed and the runtime support provides facilities to access these as well but, these are kind of outside the preview of our lecture; so we are not going to discuss, how to access printers and file systems but, we will stick to the other task that we mention memory management and so on and so forth.

(Refer Slide Time: 29:10)



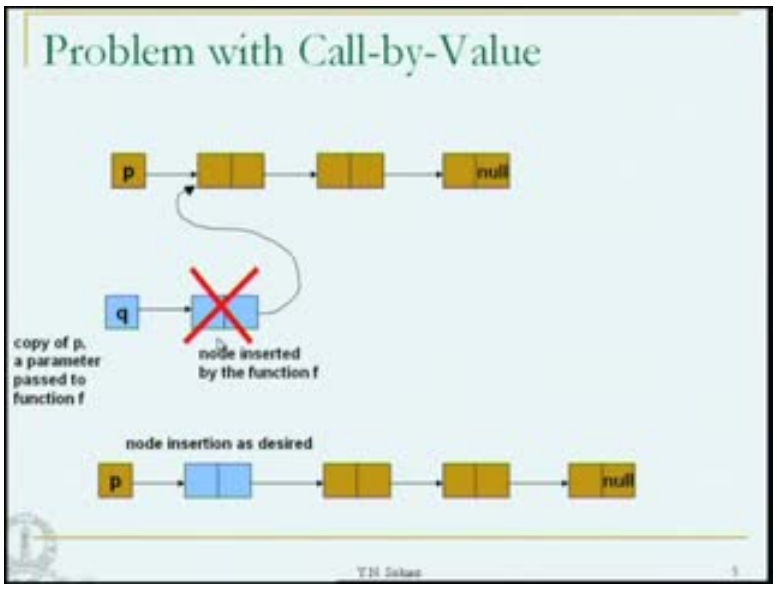
So these are some of the main task of a runtime support system. So let us review the parameter passing methods. One of the most well known parameter passing method is the call-by-value method; at runtime prior to the call, the parameter is evaluated and its

actual value is placed in a location private to the called procedure. So this is exactly what happens during call-by-value. Therefore, it is just not possible for us to change the actual parameters, we always operate on the copy. This type of parameter passing mechanism is found in C and C plus plus of course, it is also found in Pascal but, Pascal is not used anymore, so that is the reason I mentioned C and C plus plus.

C has only call-by-value method of passing parameters; it does not have other methods such as call-by-reference, we will see those methods later. So for example, even though we pass parameters frequently has a two functions, passing pointers as parameters does not constitute call-by-reference. So there is a general misunderstanding that if we pass pointers then it constitutes call-by-reference it is not.

I will show you an example, why this is not possible in C. Pointers are also going to be copied; if we pass a parameter pointer as a parameter to a C function, the pointer is also going to be copied to another location and then that copy is used. Therefore in C, there is no way to write a function which inserts a node at the front of a linked list just after the header without using pointers to pointers - that is double pointers. We will see this in the picture here.

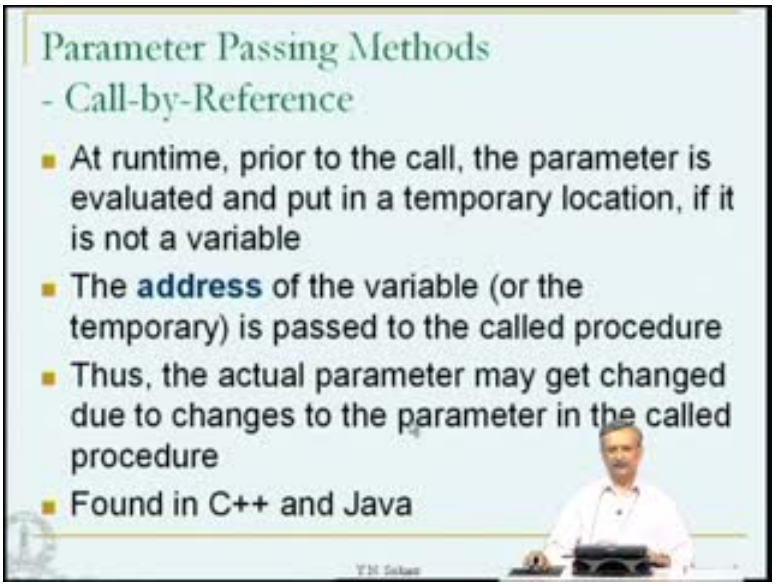
(Refer Slide Time: 31:20)



So, here is the original linked list, we have written a function - let say we assumed that we have a function, which is passed this header as a parameter; this is a pointer to this linked list. So when we make that function insert a node in the front of the header, what

we want is something like this. So this is the original header, then we have the new node which is inserted and then the rest of the linked list but, because call-by-value copies this pointer to a private location, so this is actually a copy of the original pointer - the original header. So the insert function actually inserts the node right here and then makes it point to this. So, we have not achieved this; but, this is a completely different linked list, the original linked list is unchanged, this is not what we want. So, that is the reason I said you in C you cannot do something like this, without using double pointers.

(Refer Slide Time: 32:28)

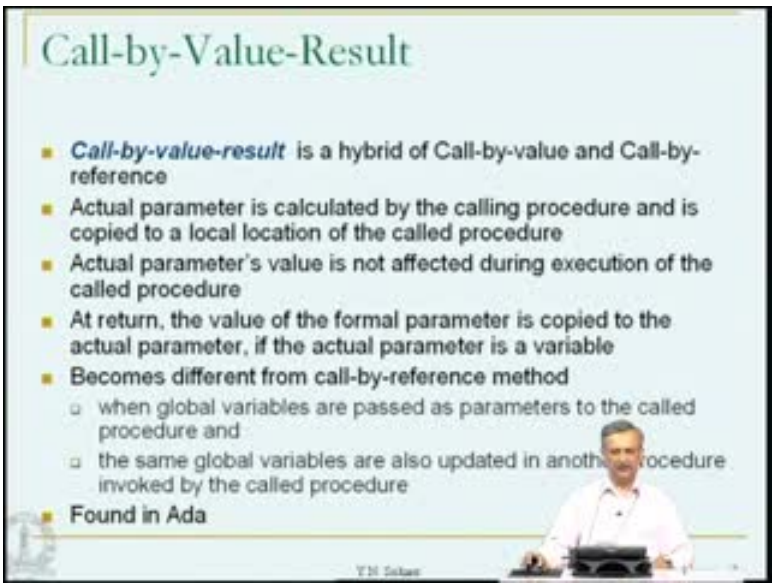


So that is about call-by-value nothing more to it than this. Let us look at call-by-reference, at runtime prior to the call, the parameter is evaluated put in a temporary location, if it is not a variable, so the address of the variable is also passed to called procedure, so this is the difference. So we have the address of the variable which corresponds to the parameter and value is also available through this particular address, so if the parameter itself is a function, there is not much that we can do - what I have said here happens - if it is the parameter is evaluated and put in a temporary location; if it is not a variable, it is an expression, so the address of this variable is passed, this temporary variable is passed but, it is not of much use.

But, if we had only one variable as a parameter then the address of that variable itself is passed to the called procedure. Thus the actual parameter may get changed due to changes to the parameter in the called procedure, such a mechanism is found in C plus

plus and Java, it is not found in C.

(Refer Slide Time: 33:56)

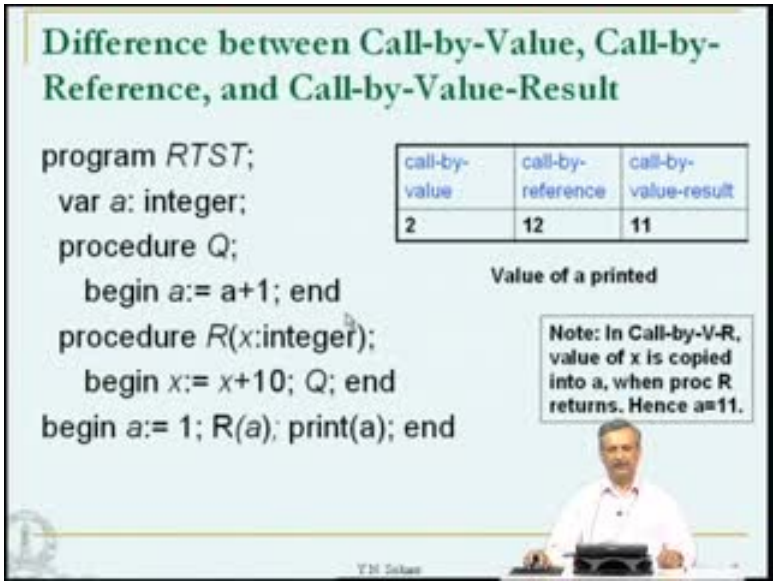


The third method of passing parameters, we look at an example after we finish call-by-value result; the third method is called call-by-value result. This is a combination or hybrid of call-by-value and call-by-reference. So what happens here is that the actual parameter is calculated by the calling procedure; it is copied to a local location of the called procedure, the actual parameters value is not affected during execution of the procedure, so far it is very similar to call-by-value. At return time, the value of the formal parameter is copied to the actual parameter; if the actual parameter is a variable.

So, if it is an expression- if the actual parameter is an expression, then you know there is not much that happens, because it would have been evaluated into a temporary and this final value is also copied to the temporary. So, the original parameter does not change but, at this point it is different; if it is a single variable, it becomes very similar to call-by-reference; the original parameter which was passed now, changes and gets the final value of the parameter that was assigned right now.

But, it is also different from the call-by-reference method when? When global variables are passed as parameters to the called procedure and the same global variables are also updated in another procedure invoked by the called procedure. I will give you an example of this and this mechanism is found in Ada.

(Refer Slide Time: 35:48)



So, here is an example to show how call-by-value result is different. A very simple program in Pascal style and since we are assuming different types of parameter passing mechanisms; it is better to use a different programming language construct to write down this rather than C or C plus plus. So there is a main program statement and a variable declaration a as integer; inside this is a procedure, which simply increments a and there is another procedure R which carries a parameter x, which is an integer; in which we have x equal to x plus 10 and then there is a call to the procedure Q and it ends. So in the main program, we initialize a to 1, call R with a and finally, print the value of a.

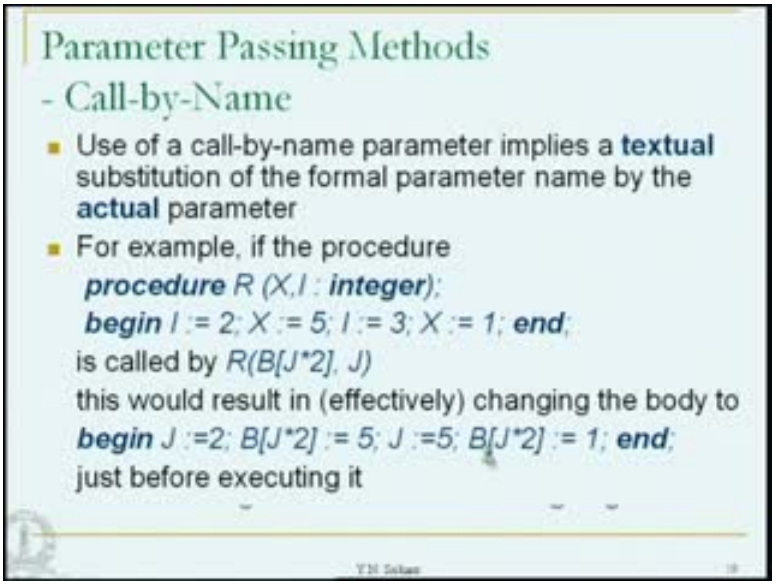
So, let us see the result for various types of parameter passing mechanism as for as, a procedure R is concerned. Let us take call-by-value first, so we have a equal to 1 here, we called R with a. So the value of a copied to a local variable that is left side, this is x itself, so x is incremented by 10, so it becomes 11; then Q is called, so Q increments a which is actually the global variable a and has nothing to do with x; x is a copy of a, as for as R is concerned; so a becomes 2, this is a global variable. So when we print a, the value of 2 gets printed (Refer Slide Time: 37:20).

Let us see what happens in call-by-reference. So again a is 1, the address a is passed to R, so this x and this a are the same. So, when we increment x by 10, x actually has - a itself has becomes 11. Even though it says x equal to x plus 1, in effect is a equal a plus 10, so it has become 11, a has become 11. You call Q, a is further incremented by

another 1 and therefore, a now becomes 12 and finally, when we print a, the value of 12 is printed out.

So this is call-by-reference; so the original itself has changed. Let us look at call-by-value result. Again we have a equal to 1, call R with the parameter a, so a is copied to the local variable x, so x becomes x plus 10, so that is 11 but, so far we are within the procedure R. Therefore, this x and this a are different, only the initial value has been copied to this x. Now we call Q. So Q actually increments the variable a, so a becomes a plus 1, which is 2; a had 1 here and a became 2 here, this x was corresponding to the local variable itself but, then once R terminates, the value of x which corresponds to this parameter is now copied into the actual parameter which is nothing but a. So, a now gets the value 11, which was the value of x, so when we say print a, it actually prints 11. So all these three methods of passing parameters actually yielded different results, because there was a global variable involved and of course, the method of passing parameters itself is very different.

(Refer Slide Time: 39:22)



Let us look at a very old parameter passing method, which is not very often used any more in languages such as C and C plus plus but, it is definitely used in function programming languages, call-by-name mechanism. In some sense a very weird parameter passing method and let see why.

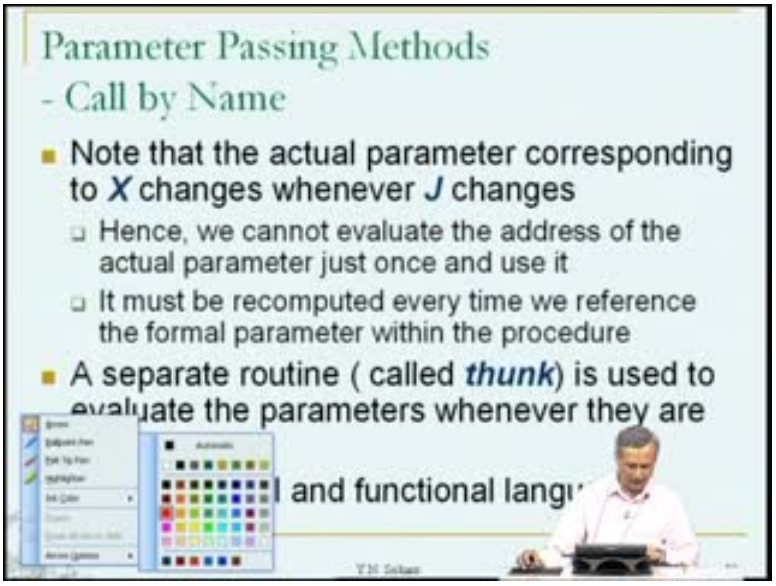
Use of a call-by-name parameter implies a textual substitution of the formal parameter

name by the actual parameter. So for example, you have a procedure R here; so R has two parameters X and I, which are integers. Let us assume that these are call-by-name parameters.

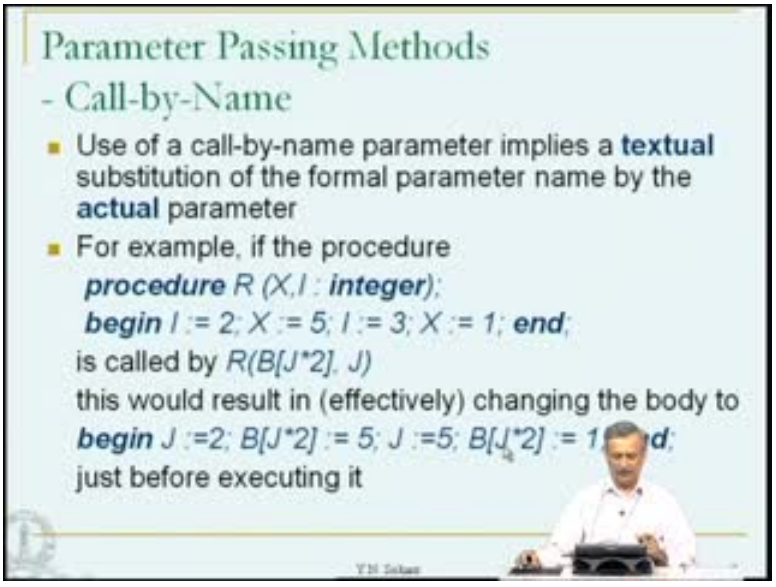
Now, inside the procedure R, we assign I equal to 2, X equal to 5, I equal to 3, X equal to 1 end and this procedure R is called by R, with a parameter of B of J star 2 and J. The result of this would be effectively changing the body to - this is I equal to 2. So I corresponds to J here, so textual substitution means J equal to 2, I has been replaced by J and then X equal to 5, the textual substitution corresponds to B of J star 2 equal to 5 because X is nothing but, B of J star 2.

Now, J had the value 2, so this became B of 4 equal to 5. Now, J equal to 5, so we had I equal to 3 here, so this J sorry this is J equal to 3, so I equal to 3 here and therefore, we also have J equal to 3 here this is not 5 and then we have X equal to 1, so X is nothing but, B of J star 2. This is the textual substitution, so because J has picked up the value 3 from here, J star 2 now has the value 6. So B of 6 gets the value 1, so even though it is the same parameter B of J star 2, with call-by-name here we assign B of 4 equal to 5 and then here we have assign B of 6 equal to 1. With call-by-reference parameter passing method this would not have happened. We would have evaluated J star 2 once and then taken the address of B J of J star 2 and pass that to this particular procedure. So this x would have been the same location in the array B irrespective of what the value of J is. In call-by-name mechanism, it is not so it is very different.

(Refer Slide Time: 42:24)



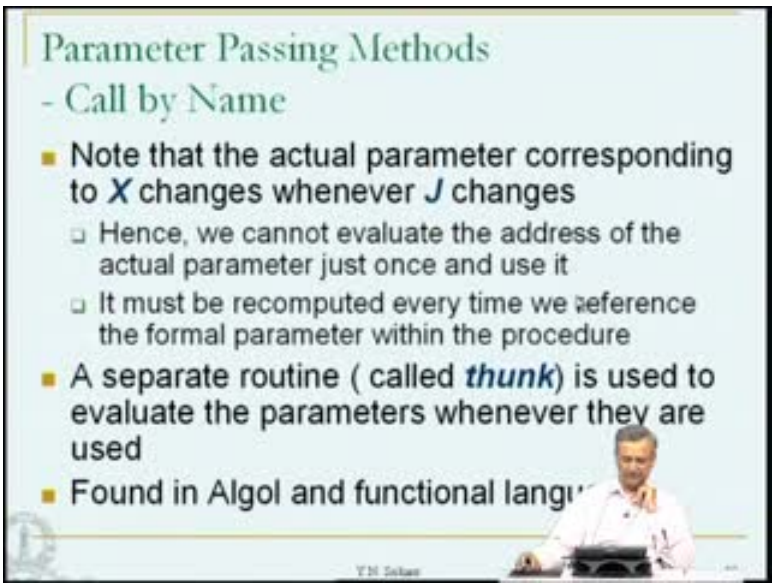
(Refer Slide Time: 43:17)



The actual parameter corresponding to X changes whenever J changes. Hence, we cannot evaluate the address of the actual parameter just once and use it; that is what we would have done in call-by-value and call-by-reference. It must be recomputed every time. We reference the formal parameter within the procedure, so that is what I mean here, so what really happens is, we will have to evaluate that particular parameter again and again. So, a separate routine called thunk is used to evaluate the parameters, whenever they are really use. So for example, let us go one step before, here is X and this is I, so B of J star 2 is evaluated again and again and the old value is not used anymore. So, it must be

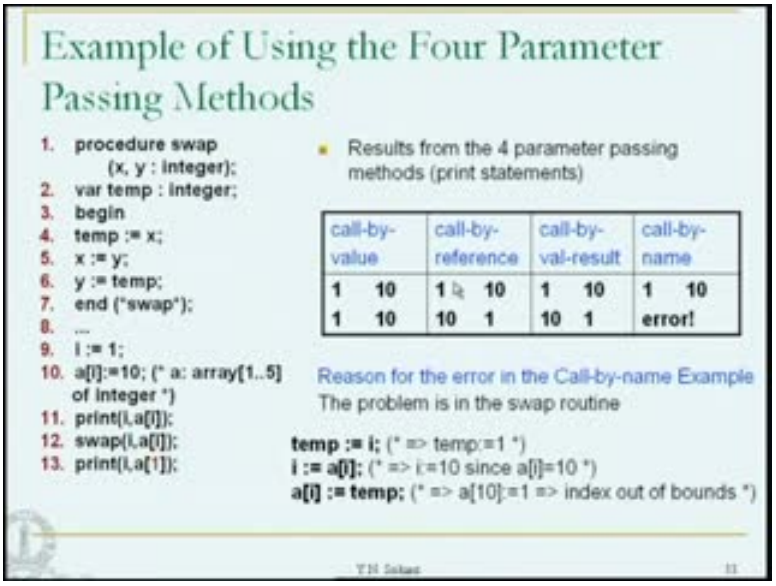
recomputed every time we reference the formal parameter within the procedure.

(Refer Slide Time: 43:31)



So, this evaluation of the parameter again and again whenever we require, it is done by a separate routine called thunk. So thunk is actually a separate procedure; so this procedure is to be generated by the compiler and it is called whenever the parameter is supposed to be accessed. So the thunk evaluates the parameter and then the assignment to that parameter or use of that parameter happens. So this was found in the old language Algol and it is still found in some of the functional programming languages, so that is the reason we studied it here.

(Refer Slide Time: 44:21)



Now, let us look at all the 4 parameter passing methods and see how they compare, so here is a simple program called swap which says temp equal to x, x equal to y, y equal to temp. So this is well known to everybody. So here we have I equal to 1 and then we have a i equal to 10, so that means a of 1 equal to 10 that is what happened and a is an array of just 5 integers. So then we have print of I comma a i, the routine swap is called with I comma a i and again, we print I of a comma 1; it is not a of i, it is a of 1 that is what is being printed.

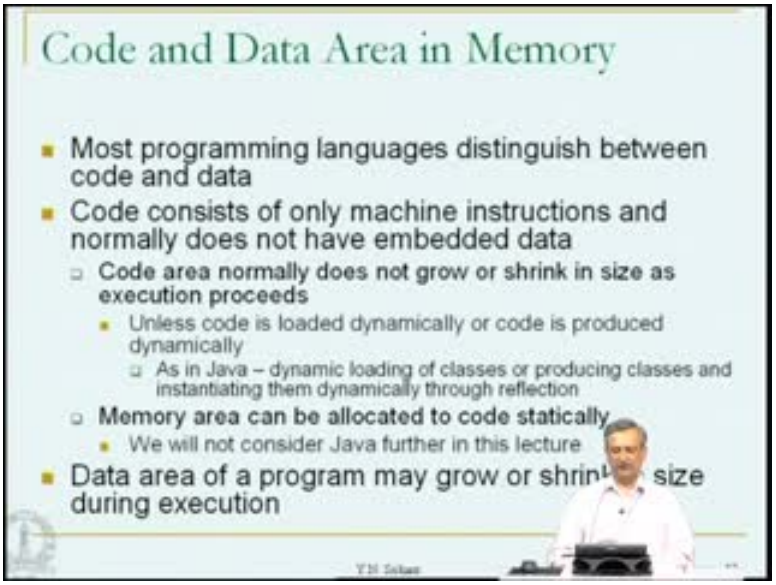
So, let us see what happens during call-by-value. So I is 1, so 1 is printed, a of 1 is also 10, so it is printed. Swap cannot be done in call-by-value because only copies will get exchanged here, the originals will not get exchanged. So, 1 comma 10 is printed again in the second print statement as well because a 1 is still 10.

Let us look at call-by-reference. So print of I comma a i prints 1 comma 10, no problem so far. Then swap, actually swaps these two values. So I becomes 10 and a of i which is a 1 becomes 1. So, what is printed here is 10 comma 1. When you look at value result it is the same, it is just that the variables are copied back to locations. So otherwise there is no difference between call-by-reference and call-by-value result in this particular program, it prints 1 10 and 10 comma 1.

Call-by-name is different, first time it prints 1 comma 10 but, then the second call after swap, the swap routine itself issues an error why? Let us look at the inside of the swap

routine by textual substitution. So temp equal to x becomes temp equal to y because x is i and x equal to y becomes i equal to a i. So now, i equal to 10, since a i equal to 10 and now when we access a i, i is 10, so we are accessing a of 10 which is an error, the array is supposed to be only 5 locations long, so index out of bounds error is issued and the program terminates. So this brings out hopefully the difference between different types of parameter passing mechanisms.

(Refer Slide Time: 47:09)



Now, let us move on to code and data area in memory. Most programming languages distinguish between code and data; in other words, the runtime system will understand, what is code and what is data. The compiler also understands, what is code and what is data. Code is placed in some part of the memory and within that code, we never place any data, at least the complier will not do it. So code consists of only machine instructions and normally does not have any embedded data.

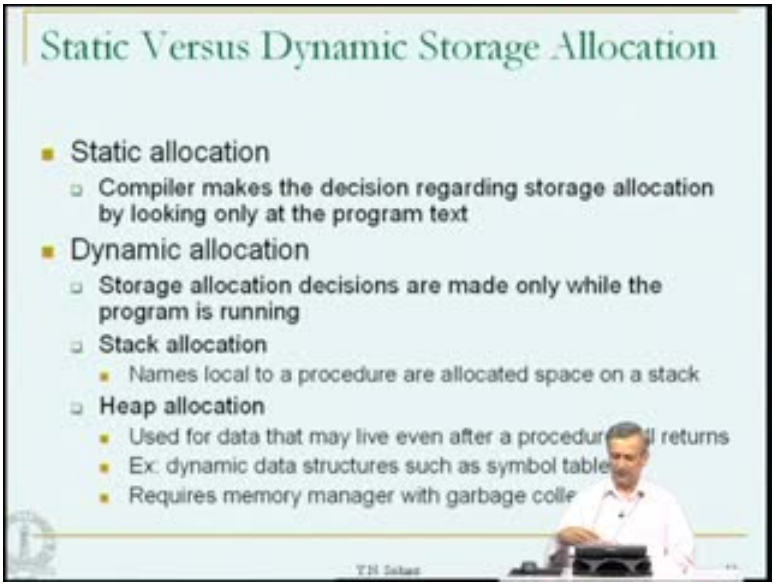
So, the code area does not grow or shrink in size as execution progresses. Thus there is very little to change the code, there is no data, it cannot expand, it cannot shrink nothing changes in between but, there are exceptions. For example, if you consider java; java has this mechanism of loading classes dynamically whenever required; classes contain basically the code for it contains a lot of code and also has lots of data.

This is an example, where I just said code or the area does not normally shrink or grow in size is not valid. In java, because of dynamic loading of classes, the code area can

grow and it can also shrink. The class say that classes just abundant once the there is no use for it. There is something even deeper. Java has this reflection interface and using that we can actually produce code through our programs and even execute that code through this reflection interface.

So, in other words not only we have the feature of dynamic loading classes; we have the feature of producing classes, which were never available anywhere, not to be seen at all and then, we can also execute this particular code; we can instant produce objects of that particular class and then, we can execute the methods of that class and so on and so forth. But, we are not going to consider java further in this particular lecture. We will assume that memory area can be allocated to code in a static manner it does not change but, data area of a program may grow or shrink in size during execution.

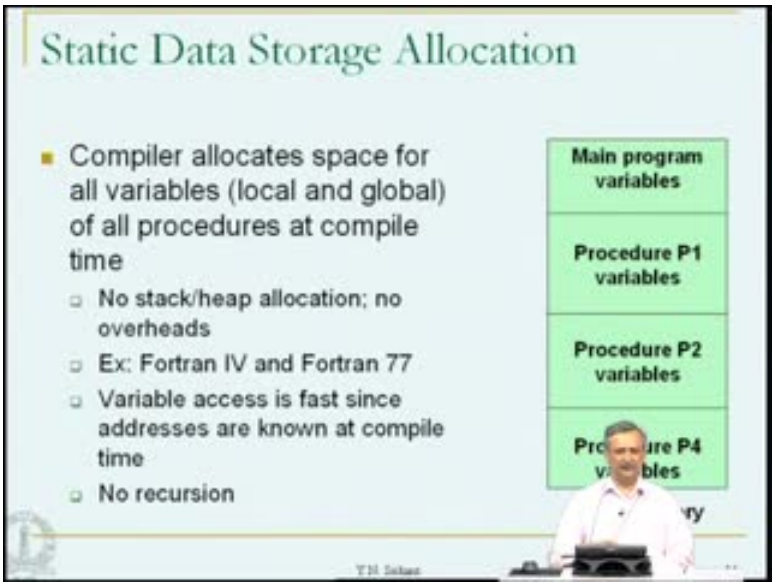
(Refer Slide Time: 50:06)



There are two types of allocations possible for memory: one is the static allocation the other is dynamic allocation. During static allocation, the complier makes the decision regarding a storage allocation by looking only at the program text; there is no execution of the code; there is no decision to be made at the time of program execution. Whereas, in dynamic allocation, storage allocation decisions are made only when the program is running. For example, there could be some stack allocation names local to a procedure are allocated space on a stack, so this happens in many languages. We will see this shortly.

There may be heap allocation for example, used for data that may live even after a procedure call returns. Dynamic data structure such as symbol tables and these require a memory manager and sometimes, they also require what is known as garbage collection, because if that dynamic data structure, say the symbol table is not used anymore and the programmer does not do anything about it the runtime system will have to run a program called garbage collection and reclaim the storage otherwise, we are going to run out of storage very soon.

(Refer Slide Time: 51:35)

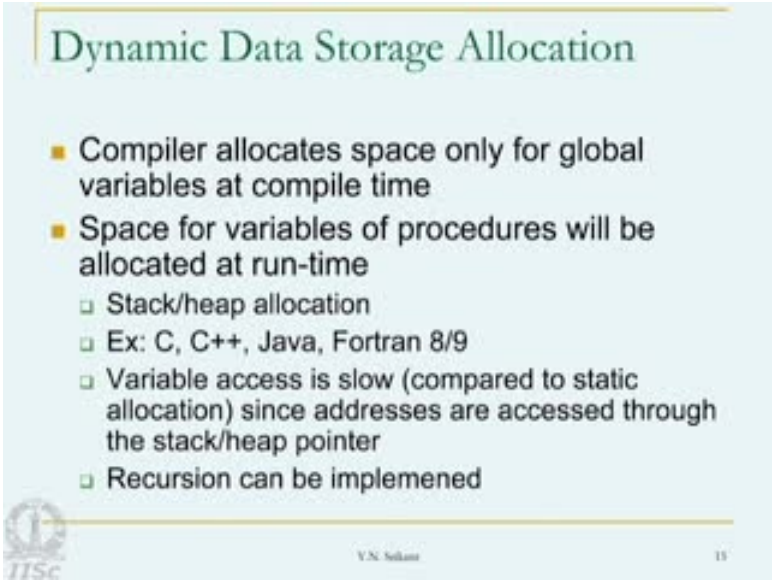


Static storage allocation an example is here, Fortran IV and Fortran 77 have this type of static data storage allocation. The compiler allocates space for all variables, see Fortran has local variables, it has global variables. So for all such variables of all procedures the storage is allocated at compile time by the compiler itself. So here is an example, there are main program variables, procedure P1 variables, procedure P2, variables procedure P4 variables etcetera.

The code area of course, is always static and it is in some other part of the memory, so that I already mentioned. Here, there is no stack or heap allocation; there are no overheads which are actually related to stack and heap allocation. Variable access is very fast, because addresses are known at compile time. There is no need to determine the address of a variable by adding various pointers and so on and so forth. The pity is such a scheme cannot support recursion. The reason is recursion requires different locations for

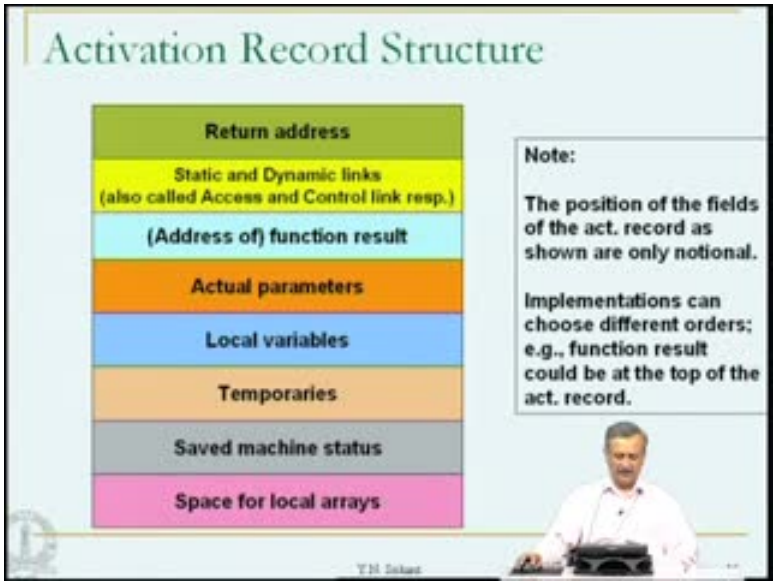
the same variable during its instantiation whereas, in static allocation this does not happen.

(Refer Slide Time: 53:05)



What is dynamic storage allocation? Compiler allocates space only for global variables at compile time. These are in a static area but, space for variables of all procedures will be allocated at runtime. This is exactly what happens in C, C plus plus, Java, Fortran 8 or Fortran 9. These are all languages were stack or heap allocation necessary for the variables. Variable access is slow compared to static allocation, since addresses are accessed through the stack heap pointer. We have to add something to that stack heap pointer the offset of that variable and then take the contents of that particular location and get the value of that variable. But, the advantage with this scheme is that recursion can be implemented. So, since we are going to have different spaces for different instances of the same procedure, it is possible to implement recursion here.

(Refer Slide Time: 54:10)



The unit of space allocation in dynamic storage allocation and in programming languages such as C, C plus plus, java etcetera is the activation record. So we are going to use this information later on. Let us understand the activation record structure. First of all, let us look at the information that is stored for each procedure, it stores the return address that is, where should it go once this procedure is completed. It stores, what are known as static and dynamic links. These are also called access and control links. Dynamic link is used to control the stack structure and the static link is used for the access of global variables.

Function result or the address of the function result is also stored on this activation record. Actual parameters which are passed to this particular procedure call are stored here. The caller will evaluate the parameters and then depending on the type of parameter passing, either the value or the address is placed in the actual parameter list. Local variables of the procedure find a place here, then we require temporaries to evaluate large expression and so on and so forth. Those are all allocated space here.

The machine status for example, the caller would be using some registers and before calling this particular procedure, it would have saved its registers but, this procedure possibly calls some other procedure and then it has to store its register contents in some place. So saved machine status is the place, where it stores the registers before calling the next procedure.

Then there is space for local arrays, even though these local variables among themselves can host these arrays; it do not do that normally, we actually locate these array space at the end, just for the sake of uniformity and otherwise, there is nothing very special. Please note that the order in which these have been placed here is not sacrosanct; it is possible that implementations choose different orders than what is indicated here. For example, function result could be at the top and then below, it could be the return result or the static dynamic link etcetera.

With this, we will close this lecture and in the next lecture, we will take up variable storage offset computation which is one of the very important jobs that a compiler has to do. It has to find offsets for various variables; where exactly are they going to be placed inside the stack, etcetera. Thank you.