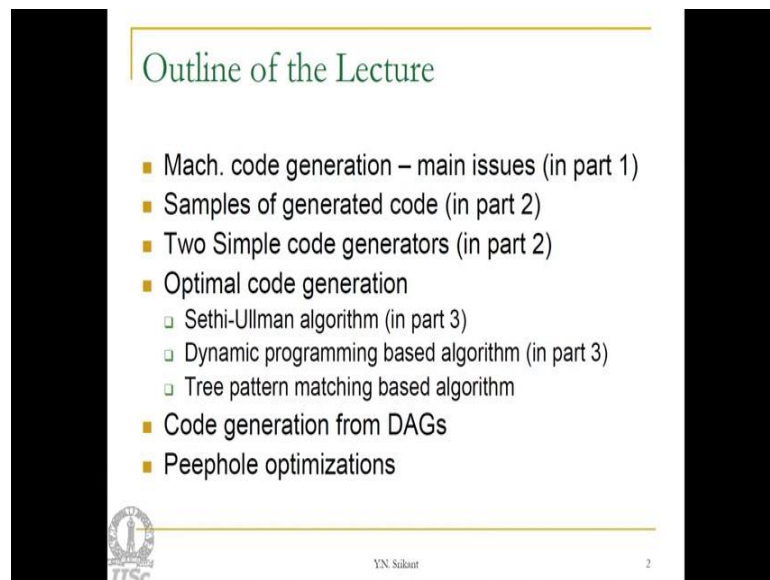


**Principle of Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

**Module - 8**  
**Lecture - 27**  
**Machine code generation Part-4**  
**Implementing object-oriented languages Part-1**

Welcome to part 4 of the lecture on machine code generation. Today we will continue with our discussion on machine code generation.

(Refer Slide Time: 00:30)

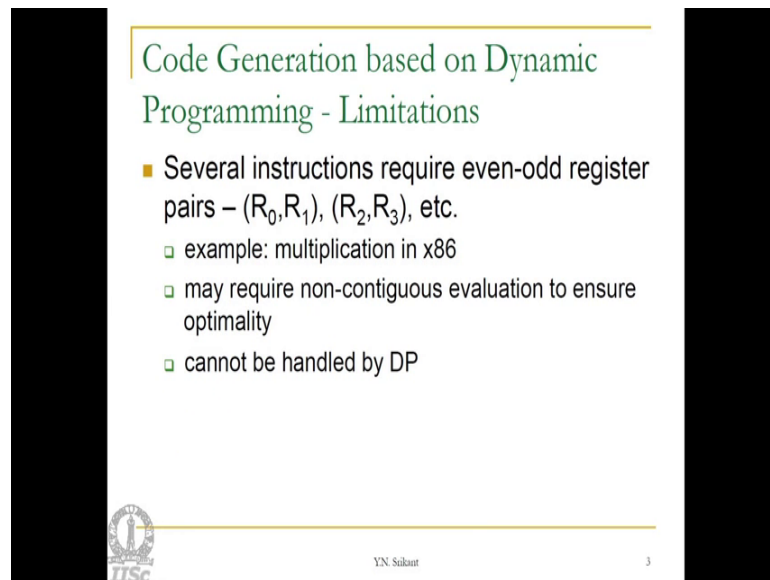


The slide is titled "Outline of the Lecture" and contains a bulleted list of topics. The list includes: Mach. code generation – main issues (in part 1), Samples of generated code (in part 2), Two Simple code generators (in part 2), Optimal code generation (with sub-points: Sethi-Ullman algorithm (in part 3), Dynamic programming based algorithm (in part 3), and Tree pattern matching based algorithm), Code generation from DAGs, and Peephole optimizations. The slide also features the IITSC logo and the name "YN. Srikant" at the bottom.

- Mach. code generation – main issues (in part 1)
- Samples of generated code (in part 2)
- Two Simple code generators (in part 2)
- Optimal code generation
  - Sethi-Ullman algorithm (in part 3)
  - Dynamic programming based algorithm (in part 3)
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

So, specially tree rewriting code generation from direct acyclic graphs and peephole optimization.

(Refer Slide Time: 00:37)



The slide is titled "Code Generation based on Dynamic Programming - Limitations" in green text. It features a bulleted list of limitations. The first bullet point is a square followed by the text "Several instructions require even-odd register pairs – (R<sub>0</sub>,R<sub>1</sub>), (R<sub>2</sub>,R<sub>3</sub>), etc." Below this are three sub-bullets, each preceded by a square: "example: multiplication in x86", "may require non-contiguous evaluation to ensure optimality", and "cannot be handled by DP". At the bottom left is the IITSC logo, and at the bottom center is the text "YN Sankar". A small number "3" is visible at the bottom right of the slide content area.


- Several instructions require even-odd register pairs – (R<sub>0</sub>,R<sub>1</sub>), (R<sub>2</sub>,R<sub>3</sub>), etc.
  - example: multiplication in x86
  - may require non-contiguous evaluation to ensure optimality
  - cannot be handled by DP

So, first of all let us see the difficulties, deficiencies their limitation of dynamic programming based code generation. The major problem is many instructions such as the multiply instruction. So, you know many other floating point instruction etcetera in modern day processors require even odd register pairs for their operation for example R 0, R 1, R 2, R 3, etcetera. They really cannot do with pairs such as R 1 R 2 or R 3 R 4, and so on. So in such cases they may require noncontiguous you know evaluation orders to become optimal code generators. So, this cannot be handled by dynamic programming as we have already studied it, it requires ad hock changes and this is one of the major limitation of the approach.

(Refer Slide Time: 01:39)

### Code Generation by Tree Rewriting

- Caters to complex instruction sets and very general machine models
- Can produce locally optimal code (basic block level)
- Non-contiguous evaluation orders are possible without sacrificing optimality
- Easily retargetable to different machines
- Automatic generation from specifications is possible

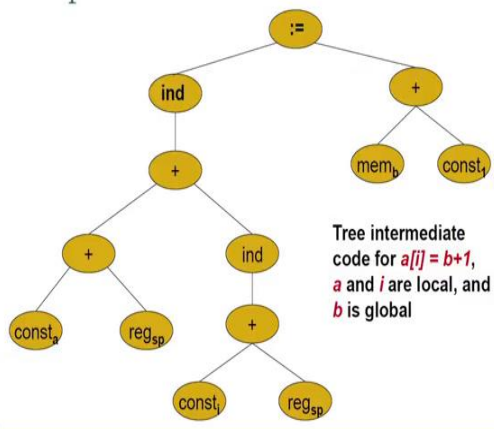


YN Sikant 4


So, to overcome this approach there has been you know a trial with tree rewriting, this is very successful it gets us to very complex instruction sets and very general machine models it can produce locally optimal code for basic blocks. Then noncontiguous evaluation orders can also be handled without sacrificing any optimality, it is of course. So, you know possible to retarget the code generator to different kinds of machines and automatic generation of code generators from specifications is also possible.

(Refer Slide Time: 02:22)

### Example



Tree intermediate code for  $a[i][j] = b + 1$ ,  $a$  and  $i$  are local, and  $b$  is global



YN Sikant 5

So, let us take an example, this is an example of you know the tree intermediate code, so instead of using quadruples tree rewriting systems require tree intermediate code. So, this tree rewriting system that I am going to show you a example will be a simple a complete tree rewriting system will require many more instructions and so on.

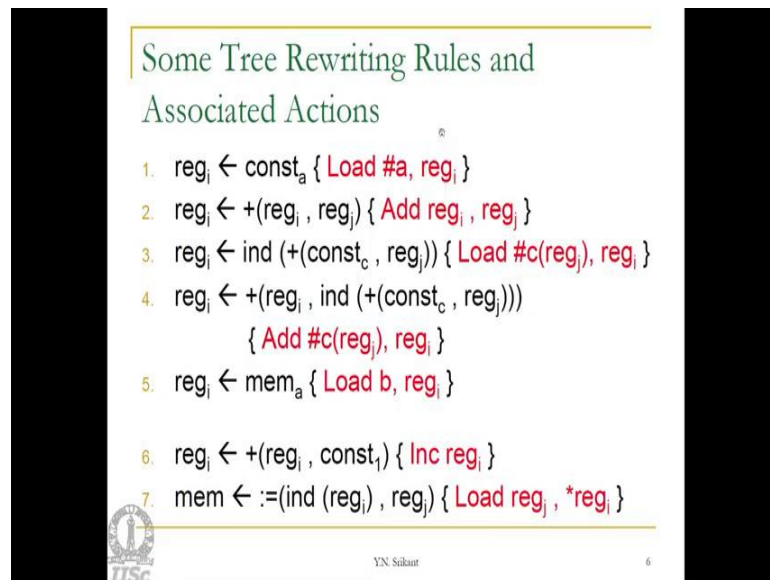
So, in this example we will take a very simple tree intermediate code, for example a  $i$  equal to  $b$  plus 1,  $a$  is a local variable it is an array  $i$  is also a local variable and  $b$  is a global variable. Now, the difference between the local and global variable is that the global variable will be present in the static area of memory. Whereas, the local variable will be accessed using the stack pointer, so that is the difference so let me explain what these operators mean and then we will go on to the tree rewriting example.

So, the first operator at the root is the assignment operator and the assignment has a left hand side and a right hand side. So, the right hand side is a very simple plus expressions and the left hand side is a an elaborate array expression corresponding to  $a$  of  $i$ . So, the right hand side first because its simpler it has a memory operand  $b$  and a constant one, so  $b$  plus 1 is synthesized here. But, on the left hand side we have the base address of the array which is a constant here right and then the register is the stack pointer, and then the contents of the stack pointer and the constant  $a$  are added.

So, that is the plus operator and, so this tells you where exactly the array is placed inside the activation record, so that is what this gives us and here we have the constant  $i$ . Then again the register stack pointer, so adding these two will tell us where the variable  $i$  is positioned in the array activation record.

So, taking the contents of that location will give you the contents of the location  $i$ , so adding that to a base address within the activation record for the array  $a$  will give us the position of the element here. So,  $\text{ind}$  operator tells us that we should really consider the address of that because it is on the left hand side of the assignment operator, so this is the meaning of this entire tree intermediate code.

(Refer Slide Time: 05:17)



Some Tree Rewriting Rules and Associated Actions

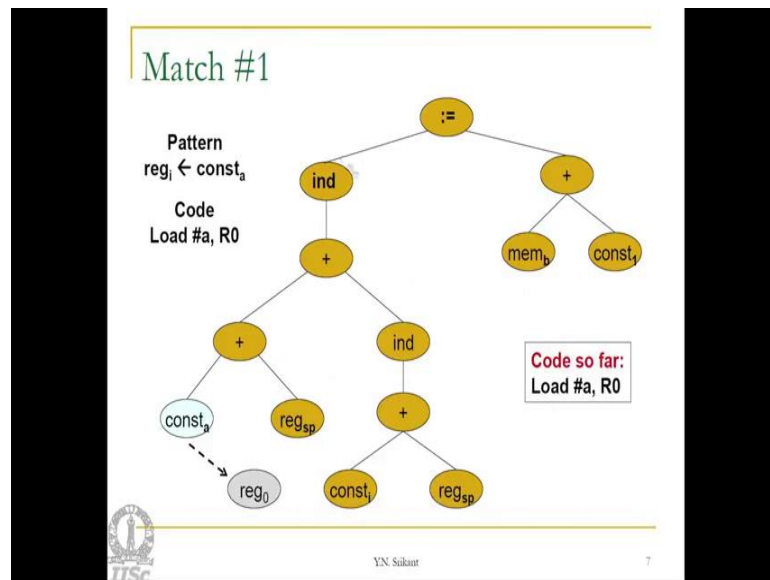
1.  $reg_i \leftarrow const_a \{ Load \#a, reg_i \}$
2.  $reg_i \leftarrow +(reg_i, reg_j) \{ Add \#reg_i, reg_j \}$
3.  $reg_i \leftarrow ind (+ (const_c, reg_j)) \{ Load \#c(reg_j), reg_i \}$
4.  $reg_i \leftarrow +(reg_i, ind (+ (const_c, reg_j))) \{ Add \#c(reg_j), reg_i \}$
5.  $reg_i \leftarrow mem_a \{ Load \#b, reg_i \}$
6.  $reg_i \leftarrow +(reg_i, const_1) \{ Inc \#reg_i \}$
7.  $mem \leftarrow :=(ind (reg_i), reg_j) \{ Load \#reg_j, *reg_i \}$

Y.N. Sankar 6

Now, let me also show you some tree rewriting rules that we are going to use in our example the first tree rewriting rule says  $reg_i$  with an arrow which is pointed towards the left and then  $const_a$ . So, remember in a context free grammar production this arrow is actually pointing the other way, whereas in a tree rewriting specification the arrow points from right to left to the way. So, you know read this would be  $const_a$  would be rewritten or replaced by  $reg_i$ , so that is going to be the semantics of this rewriting rule. So, the way the tree pattern matching or tree rewriting systems would be it determines where this  $const_a$  is present in the entire tree.

Then it tries to once it matches it replaces with node called register with an appropriate register number allocated to it. So, when this happens at appropriate time a small code piece of code will also be emitted and that code in this case will be  $load \#a, reg_i$ . So,  $\#a$  is you know immediate mode of addressing and  $reg_i$  would be the appropriate register which is used in this instruction. So, similarly there are other instructions with  $plus \#reg_i, reg_j$  and then  $ind \#c, reg_i$  etcetera, so we will understand each of these as we go on.

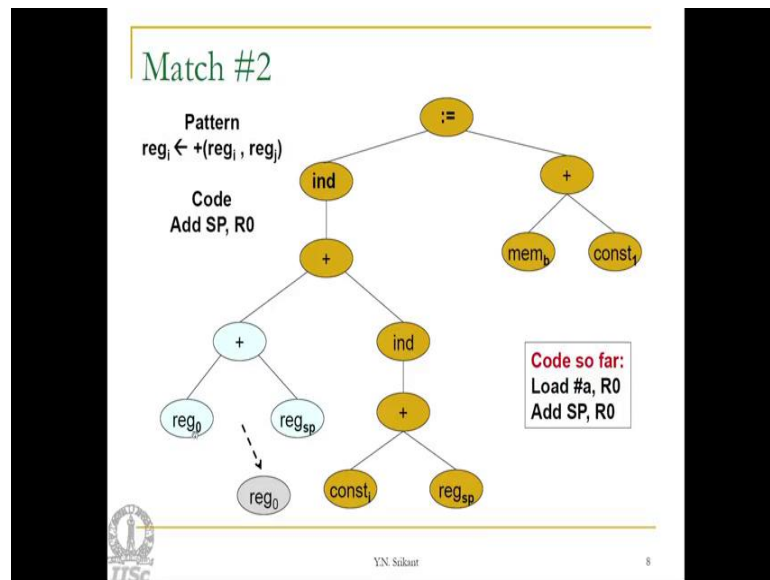
(Refer Slide Time: 06:58)



So, match number 1 in this intermediate code, so the match happens at this place this is the constant a node. So, as I already indicated to you the pattern which matches at this node is the reg i arrow const a the pattern. So, const a would be rewritten by reg and at this point there is a register location which happens and register 0 is allocated to the hold the constant a at this point.

So, this node const a would be rewritten by reg with the appropriate you know 0 register number 0 allocated to it and the code generation does not happen immediately at this time it can happen later also. But, let us assume that code generation can happen in a hand in hand with this matching itself I will tell you why it should not be run immediately. But, at this point it is ok to have it immediately the code which is emitted would be load hash a, R 0 indicating that constant a would be loaded into the register number 0, so this is the code which is generated so far.

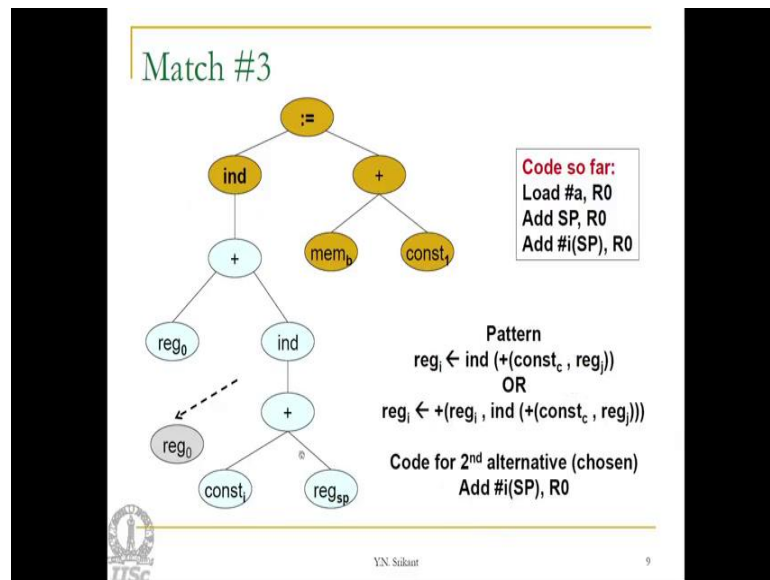
(Refer Slide Time: 08:16)



Now, so we have a register number you know const a being replaced by reg 0, now that along with the other two nodes in the tree will be constitutively next pattern which is to be matched. So, the pattern which is going to be matched at this point is the reg i left arrow plus reg i, reg j indicating that root is plus the left sub tree is reg 0 and the right sub tree is reg S P.

So, here reg i is identified as reg 0 and reg j is identified with reg S P and plus remains as it is, so when this tree pattern is reg i ced to reg 0 which is the left hand side. Now, we emit the code, add S P, R 0, so the code which is emitted so far says load hash a, R 0 add S P, R 0, so this entire sub tree will now be emitted from the intermediate code and it will be replaced by reg 0.

(Refer Slide Time: 09:19)



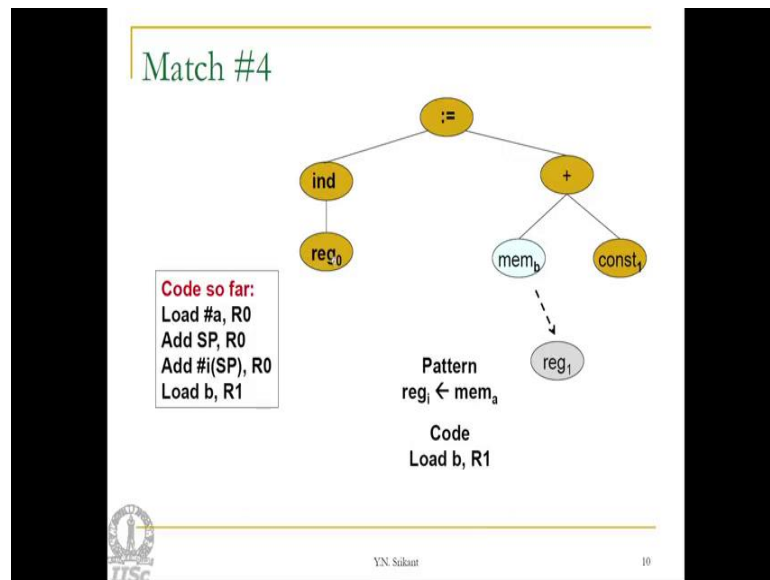
So, this is the node which we replace so far and, now we have this entire big tree which is found as the pattern in the intermediate core. So, in fact there are 2 possibilities of match at this point, one is to use a smaller tree that is this part and the second option is to use this bigger tree which is this entire thing. So, the smaller one would be, so the ind plus const c, reg j that is this part and the bigger one would be plus reg i ind plus const c, reg j which is this entire big one.

So, then what is the difference between these two matches the difference is if we actually match the bigger pattern that we have found here it leads use to you know emitting fewer number of instructions. Whereas, if we match this smaller pattern first and then go on to produce code it is possible that we emit more number of instructions and the code generation scheme may not produce optimal code.

So, in this case let us assume usually it is better to assume better to choose the bigger pattern. So, let us assume entire tree gets matched and that would be replaced by reg 0 as far as our pattern tree pattern matching rule says. So, again reg i corresponds to reg 0 and the const c corresponds to you know const i here and then reg j corresponds to reg S P, so if this is matched.

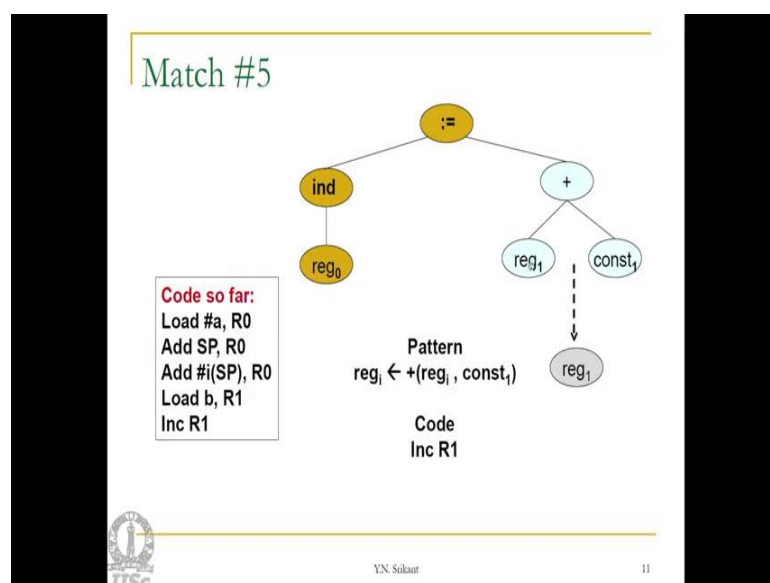


(Refer Slide Time: 11:02)



So, then you have a reg 0 here that was matched before and now the new node that matches is mem b which is going to be replaced by reg 1 the pattern which matches is reg i left arrow mem a. So, a corresponds to b here and then this one corresponds to i here and the code that is generated is load b, r 1. So, the indication is this memory is now going to be replaced by a register showing that memory has been moved into the memory. So, location has been moved into contents of the memory location have been moved into a given register at this point. So, this becomes reg 1 so far this remains as it is, so we have emitted four instructions so far.

(Refer Slide Time: 11:50)



Now, match number 5, this was the previous match that we found and now we match this entire thing. So, here it is very convenient to emit an increment instruction it is very easy to see that this is a plus this is reg and this is a const 1. So, whenever there is a constant of one it is easy to emit a increment instruction so the pattern which matches at this point is reg i comma const 1 and the associated code would be increment instruction. So, we emit the increment R 1 instruction and reduce this entire tree pattern to the reg 1, so this is the code that we have emitted so far.

(Refer Slide Time: 12:34)

### Match #6

```

graph TD
    A[":="] --- B["ind"]
    A --- C["reg1"]
    B --- D["reg0"]
    A -.- E["mem"]
        
```

Pattern

$$\text{mem} \leftarrow :=(\text{ind}(\text{reg}_i), \text{reg}_j)$$

Code

Load R1, \*R0

**Code so far:**  
 Load #a, R0  
 Add SP, R0  
 Add #(SP), R0  
 Load b, R1  
 Inc R1  
 Load R1, \*R0


YN Sikant
12

Now, what remains is this and it matches the pattern assignment in reg i, reg j and the code generated would be load R 1, star R 0 and this entire thing reduces to memory. Now, we stop at this point because there is no more rewriting that can be done, so if the instruction which is generated here says take the contents of R 1 and put it into the location 0.2 by reg 0. So, that is what star R 0 really means, so this ind on the left hand side of an assignment always says take the address and not the contents. So, the code that we have generated so far corresponds to this, so this is the general method of performing you know tree rewriting and code generation.

(Refer Slide Time: 13:28)

### Code Generator Generators (CGG)

- Based on tree pattern matching and dynamic programming
- Accept tree patterns, associated costs, and semantic actions (for register allocation and object code emission)
- Produce tree matchers that produce a cover of minimum cost
- Make two passes
  - First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost
  - Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched
- Twig, BURG, and IBURG are such CGGs

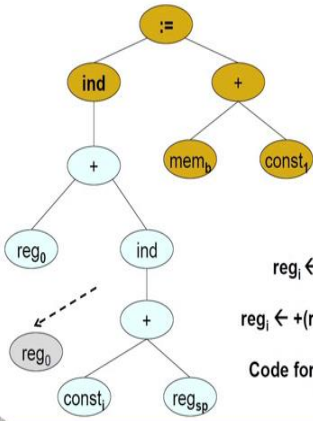


YN Sankar 13

So, now how does a code generate generator really work, so it is based on tree pattern matching and dynamic programming. So, where does dynamic programming come in here, so that is easy the reason is in many in this particular example here.

(Refer Slide Time: 13:48)


### Match #3



**Code so far:**  
Load #a, R0  
Add SP, R0  
Add #i(SP), R0

**Pattern**  
 $reg_i \leftarrow ind (+ (const_c, reg_j))$   
OR  
 $reg_i \leftarrow + (reg_j, ind (+ (const_c, reg_j)))$

**Code for 2<sup>nd</sup> alternative (chosen)**  
Add #i(SP), R0



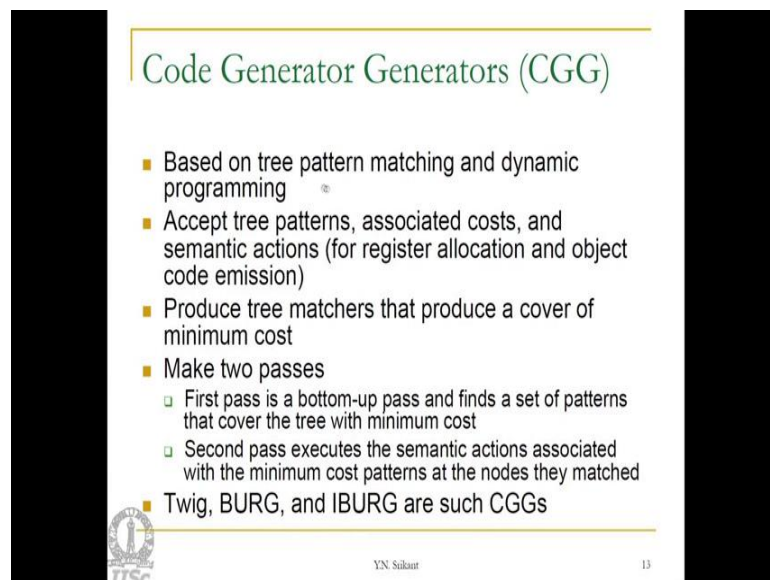
YN Sankar 9

Now, we saw that there were two alternatives for the pattern that matched here, so in general what we do in a tree rewriting system is to associate a cost for each one of the pattern matches. Then once we associate a pattern match with every node of the tree it is said to cover the tree the minimum cost cover of the tree is computed. So, from that we

deduce the patterns which match at this at the various nodes and produce the optimal cost code.

So, code emission will not happen hand in hand with tree pattern matching tree pattern matching happens. So, first the various nodes are actually decorated with the matches that the patcher has produced the cost of these matches is computed and, finally the cost of the match at the root gets computed. So, this would have happened in this case also it so happens that the second alternative gives us better code, so that code was used in our example.

(Refer Slide Time: 14:57)



The slide is titled "Code Generator Generators (CGG)" in green text. It contains a bulleted list of features:

- Based on tree pattern matching and dynamic programming
- Accept tree patterns, associated costs, and semantic actions (for register allocation and object code emission)
- Produce tree matchers that produce a cover of minimum cost
- Make two passes
  - First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost
  - Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched
- Twig, BURG, and IBURG are such CGGs

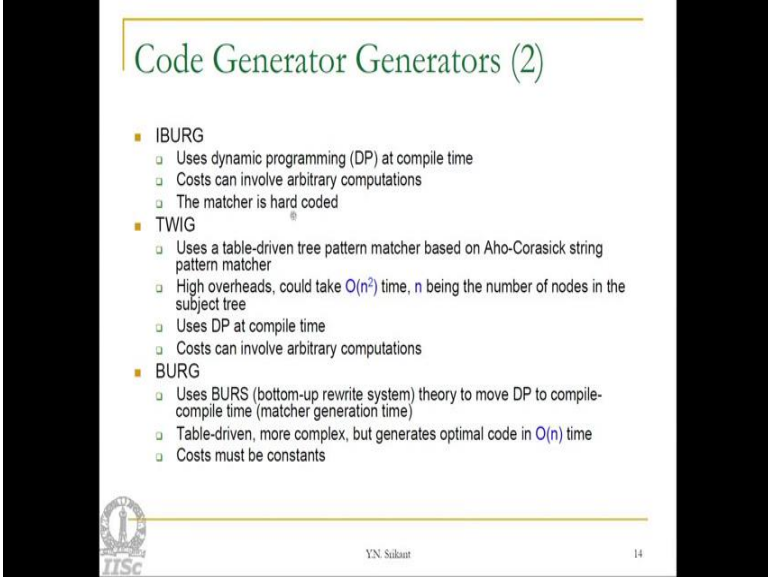
At the bottom left of the slide is the IITSC logo. At the bottom center, it says "YN Sikant". At the bottom right, it says "13".

So, that is what this code generator in general does it uses dynamic programming to compute the minimum cost cover. So, we accept tree patterns associated costs and the semantic actions associated with the tree patterns for register location and code generation. So, and taking all this as input it produces tree matchers tree pattern matchers that produce a cover of minimal costs.

So, dynamic programming becomes inbuilt into this tree pattern matching process and we actually make 2 passes the pattern match rather the code generator which is produced makes 2 passes. So, the first pass is a bottom of pass and it finds a set of patterns that cover the entire tree that I explained just a few minutes ago and once we find this, we find the minimum cost cover as well.

Then in the second pass we execute the semantic actions associated with the minimum cost cover and you know and emit the code that is necessary, so this is how the code generators which are produced by the code generator systems work. So, there are many not commercial, but very successful open source tools are available in the public domain one is called twig, the other is called burg and the second one, third one is called i burg.

(Refer Slide Time: 16:31)



The slide is titled "Code Generator Generators (2)" and lists the following details:

- IBURG
  - Uses dynamic programming (DP) at compile time
  - Costs can involve arbitrary computations
  - The matcher is hard coded
- TWIG
  - Uses a table-driven tree pattern matcher based on Aho-Corasick string pattern matcher
  - High overheads, could take  $O(n^2)$  time,  $n$  being the number of nodes in the subject tree
  - Uses DP at compile time
  - Costs can involve arbitrary computations
- BURG
  - Uses BURS (bottom-up rewrite system) theory to move DP to compile-compile time (matcher generation time)
  - Table-driven, more complex, but generates optimal code in  $O(n)$  time
  - Costs must be constants

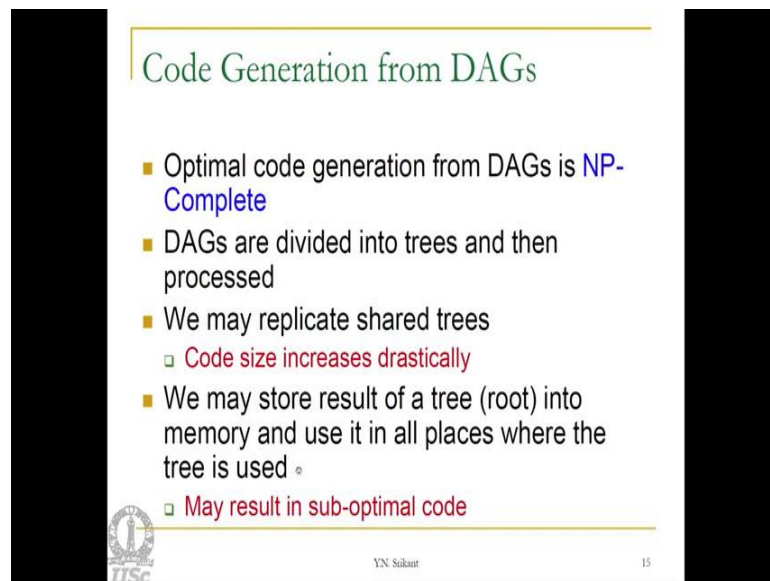
At the bottom of the slide, there is a logo for IITSC on the left, the name "YN Srikant" in the center, and the number "14" on the right.

So, let me tell you a briefly about each one of these i burg uses dynamic programming at compiled time. So, in other words when it is building the code generator it does not use any dynamic programming at all it does not analyze the you know tree rewriting rules to produce the best tree pattern matching and so on and. So, forth, but if uses dynamic programming at code generation time, therefore the advantage is the cost can be very arbitrary it can be any computation. But, it is a bit slow compared to the other schemes the matcher of course is hard coded twig uses a table driven tree pattern matcher and it uses a string pattern matching based tree pattern matcher it has very high overheads.

So, it uses  $O n$  square time and  $n$  being the number of nodes in the subject tree the it uses dynamic programming again at compile time like i burg does and of course the cost can be arbitrary burg is the best of the tools that is available. Now, again, but the problem is burgs uses bottom of tree rewriting system and the theory is extremely complicated, therefore it is very hard to produce a code generator system using this particular theory.

But, it produces optimal code you know it really moves the dynamic programming technique into the code generator time rather than the code generation time and the costs must be constants. So, the implication of this cost being constants is that you cannot have arbitrary computations, but you can only have a constants for each one of the rules that we have mentioned.

(Refer Slide Time: 18:27)



The slide is titled "Code Generation from DAGs" in green text. It contains a bulleted list of points:

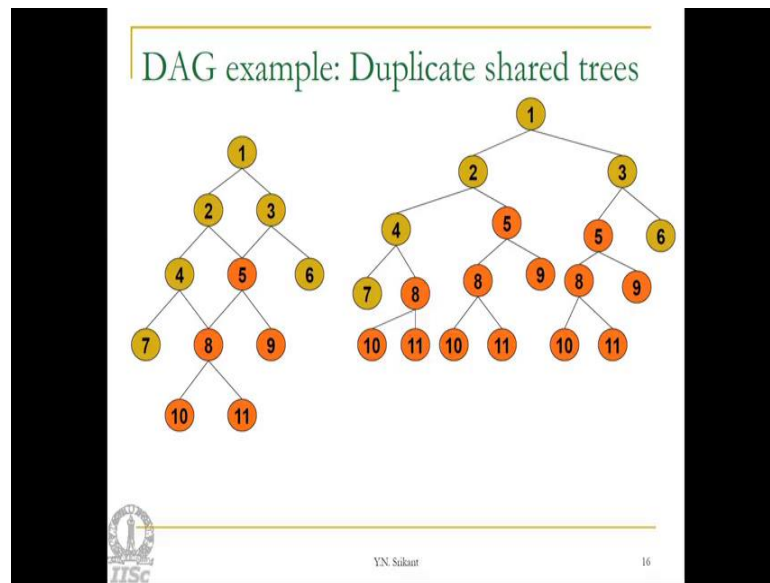
- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code

At the bottom left of the slide is the IITSC logo. At the bottom center is the name "YN Srikant". At the bottom right is the number "15".

Now, let us move on to the code generation from directed acyclic graphs, so far we have actually looked at trees and studied mechanisms to produce code generation from trees. But, in practice a basic bug gives rise to directed acyclic graphs, so code generation from directed acyclic graphs is known to be an N P complete problem. So, it cannot be done in polynomial time, it requires you know exponential amount of time to produce optimal code. So, directed acyclic graphs are divided into trees and then processed otherwise there is no algorithm which processes directed acyclic graphs and produces optimal code I am going to give you examples of how this happens.

So, we may actually replicate shared trees or we may store the result of the tree into you know memory and then use it in all the places where the tree is used. So, there are two possibilities if we share the replicated trees in such a case the code size increases you know dramatically, whereas if we store it into memory and the use it then it may result in suboptimal code.

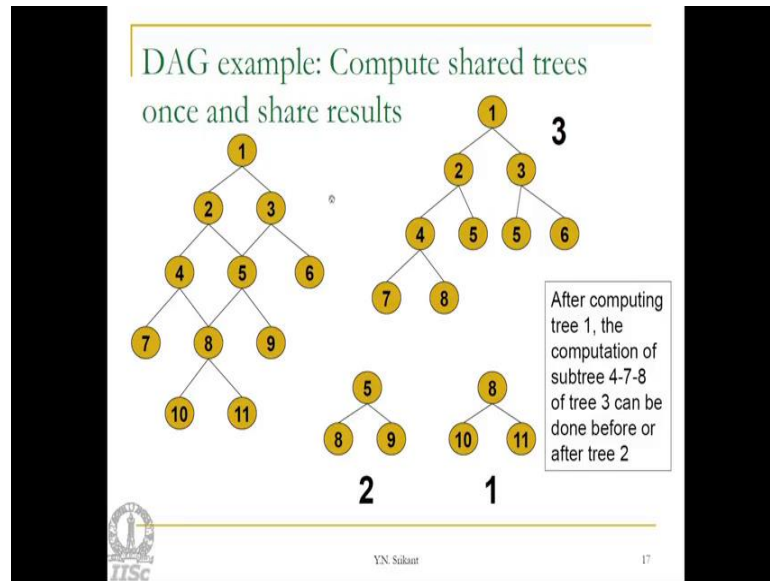
(Refer Slide Time: 19:51)



So, let me give you an example to drive home the point, now this is a dag why this is a directed acyclic graph well the nodes are shared here. So, this node and this node both of them are shared, so we cannot conduct any tree pattern matching on this directed acyclic graph. So, what we really try to do is take the shared tree, this is a shared tree, it is shared between the nodes 2 and 3, so we actually replicate the shared tree for both these nodes. So, the right sub tree of 2 is shared and the left sub tree of 3 is shared, so we replicate and make it into 2 separate copies.

But, in the meanwhile a small sub tree 8 and 11 is again shared between 4 and 5, so we really have to make a copy of this small tree for 4 as well. So, we do that here, so is easy to see that you know such way code replication increases the size of the tree too much. So, in fact this is now even though the code that is produced will be very fast the this is not a viable scheme it really increases the size of the code too much the second scheme would be you know. So, I forgot to mention that on this tree we really apply you know tree pattern matching and code generation as we have studied so far. So, that would produce code that is optimal to this tree, but since we are having replicated the optimality is lost with respect to the directed acyclic graph.

(Refer Slide Time: 21:35)



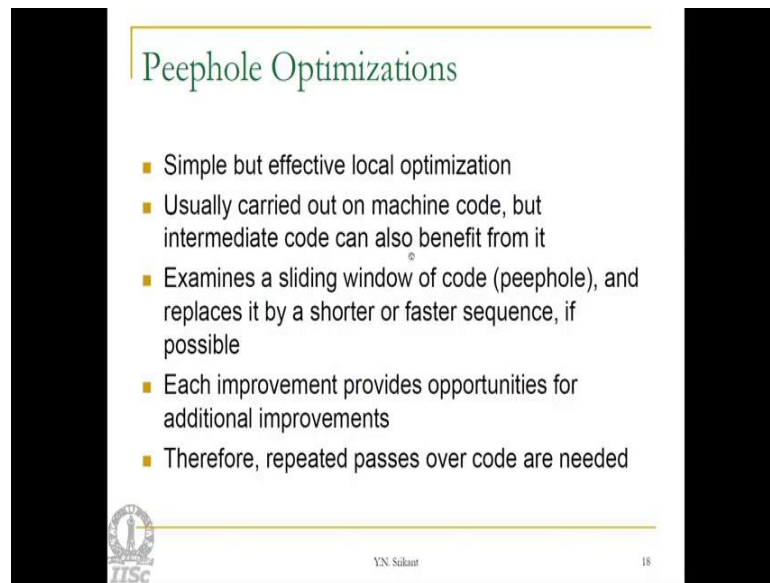
So, what happens if we you know chop the tree at this point, for example let us chop the tree at this point and let us chop the tree at this point as well. So we have produced 3 sub trees, so the first one it has 5, you know which is not shared which assumed to be a memory location. Then the second sub tree is a 5, 8 and 9 and then the third sub tree is a 8, 10 and 11, so the indication is we definitely have to evaluate 8, 10, 11 first because 8 is contained as the left sub tree of 5 here. Then it is also contained as a right sub tree of four so it is not possible to evaluate this pattern and this pattern before we evaluate this tree, so this must be evaluated first.

Now, the question is having evaluated 8 then 11, should we go ahead with the evaluation of parts O, this tree or should we evaluate this tree and then go to this. So, as you can see there are several choices here, so in general is the tree is very large and we have many such smaller trees produced out of this pruning operation. So, there will be a large number of choices which need to be checked before we decide that one of them gives us optimal code this is the difficulty.

So, the code generation strategy again becomes you know an expensive unless we use a heuristics to say this is the order in which we produce the code. So, whether we use duplication or we use sharing the code generation scheme is still really is more difficult than in the case of ordinary trees.




(Refer Slide Time: 23:30)



**Peephole Optimizations**

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

 YN Sankar 18

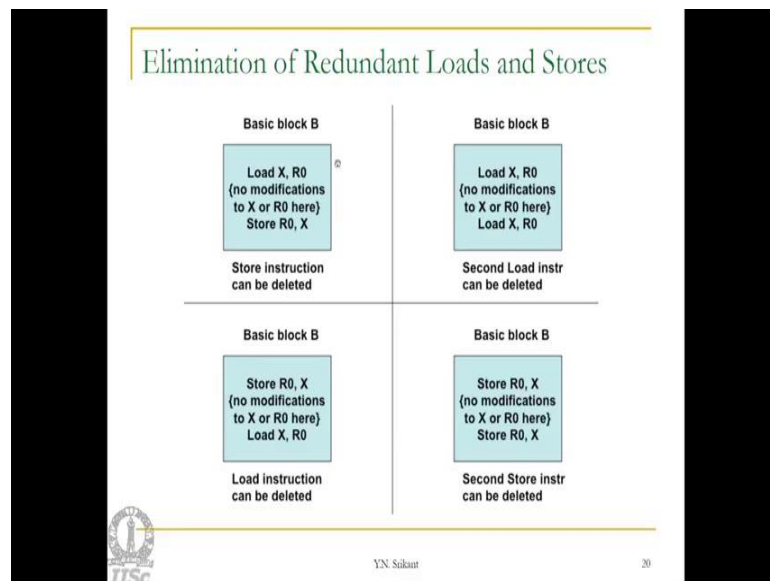
So, let us move on to a minor optimization, which is called as a peephole optimization, peephole optimization by definition looks at a small window of instructions. So, this is a very simple, but very affective local optimization and it is usually carried out on machine code. But, there is nothing to prevent you know us from doing it on intermediate code the reason we do not do it on intermediate code is that it is not that affective compared to what is done on machine code.

So, reason is simple when we generate machine code there is many more instruction for each one of the intermediate code instructions. So, there is much more scopes to produce, you know the more optimized code when we actually do peephole optimization. Whereas, if we do it on intermediate code then the opportunities for code optimization would be fewer x, now the details of peephole optimization it examines a sliding window of code.

So, this is what is known as a peephole and then it tries to find patterns within that peephole replaces the pattern which are found by more efficient code that it can actually determine. Now, the entire sequence of instructions in the window becomes possibly much more compact and efficient sequence of instructions compared to the previous one. So, improvement opportunities for additional improvements will accrue as we go on doing it for once we improve the code.

Then you know it may give you opportunities for more improvements, so we may have to repeat the peephole optimization until no more improvements are possible so peephole optimizations are many in number. Now, for example eliminating redundant instructions and then eliminating unreachable code eliminating jumps over jumps algebraic simplifications strength reduction and use of machine idioms. Thus, these are, these are this is a small list there are many other minor possibilities, so let us understand each of these by looking at an example.

(Refer Slide Time: 26:19)



So, how do we eliminate redundant load and store instructions in a peephole, so let us assume that here is a basic block it contains many instructions among these instructions. Now, there is a load E, R 0 followed by, you know many other instruction which do not modify R 0 or C and then at the end of this sequence there is a store R 0, X.

So, now if you assume that there are no modifications to X and R 0 in this space it is very clear that the second instruction store R 0, X is redundant. So, let us see why there is load X, R 0, so X and R 0, now contain the same value, so if X is not modified here low and r 0 is not modified here loading R 0 into X again will not change the value of X at all. So, again R 0 has not been modified, so there is no new value in R 0 to be put into X, the same is true or the other 3 patterns as well we have a load X, R 0.

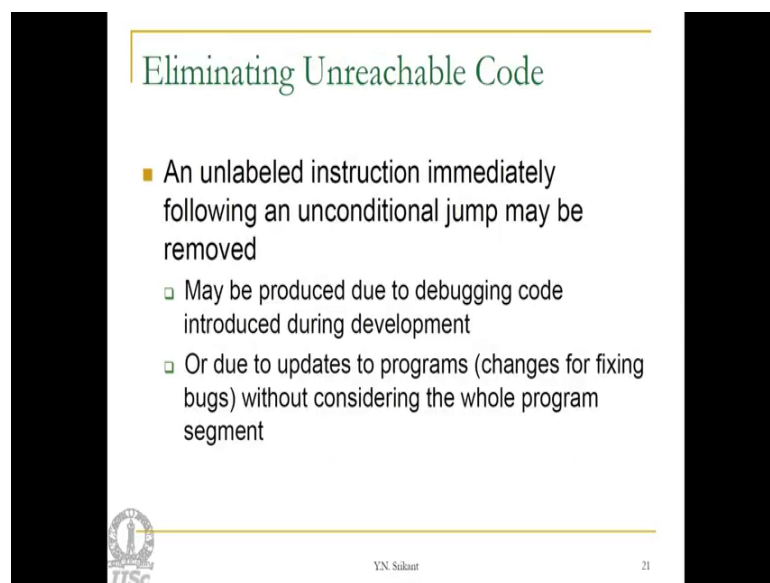
Then there are no instructions which change X either X or R 0, there is a load X, R 0, again very clearly this is redundant. So, third one store R 0, X and followed by

instructions which do not change either X or R 0. Then we have a load X, R 0 again you know since R 0 and X do not change the second one does not, put a new value into R 0. So, the load instruction is redundant the fourth pattern is store R 0, X followed by a sequence of instructions which do not change either R 0 or X.

Finally, another instruction store R 0, X again this is redundant because the store has already been executed previously. So, in any of these 4 patterns, the pattern matcher would detect that there is load followed by instructions which do not change value of either X or R 0 followed by an R 0 store R 0, X. So, what is role of peephole here the possibility is the peephole is the entire basic block which may have thousands of instructions which is a bit too large usually a peephole consists of just about 10 to 20 instructions.

So, it would examine the instructions in than peephole exhaustively for each one of the patterns that we have actually identified. So, in this case these are the 4 patterns or redundant load and store elimination if any these patterns are found. Then it does exactly what is denoted here delete these store instruction delete the load instruction etcetera. So, this is an example of how peephole optimization improves the by removing redundant load and store instructions.

(Refer Slide Time: 29:31)



The slide is titled "Eliminating Unreachable Code" in green text. It features a bulleted list with three items: a main bullet point followed by two sub-bullet points. The slide also includes a logo for IITSC in the bottom left corner, the name "YN Sankar" in the bottom center, and the number "21" in the bottom right corner.

### Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

IITSC

YN Sankar

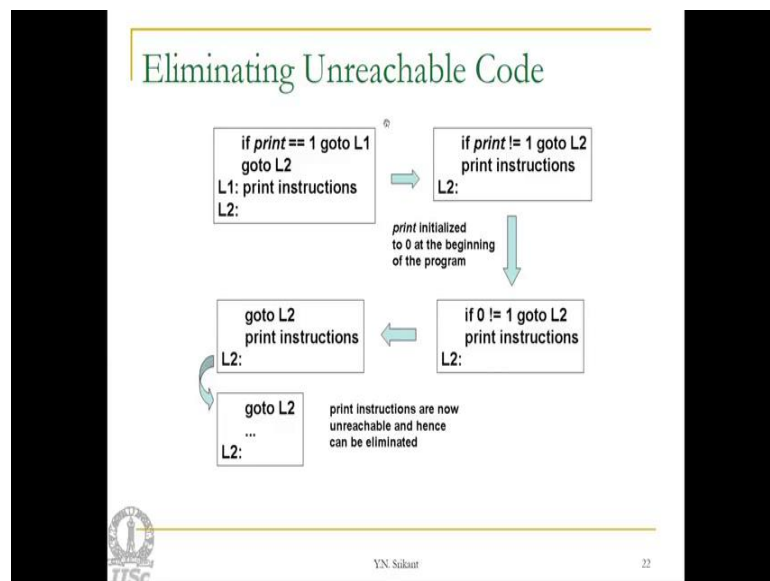
21

Now, how does one eliminate unreachable code the code that becomes unreachable may be because of many reasons. So, see if there is a, you know an unlabeled instruction

which follows an unconditional jump instruction then it is very clear that we will never get to the instruction. But, following the unconditional jump because there is no label to that instruction so such an instruction can be easily removed, but in the first place how did such a situation arise, I will give you an example.

So, it may be produced because we have debugging code introduced into the program during development stage. But, it was never removed in the final version of this software one possibility second possibility there were lots of updates which were made to the program. So, as the program changed there were many pieces of code which became you know a kind of a bad they it had unreachable code and so on. But, it was never ironed out and never improved by the programmer, so the final code again contain such unreachable code.

(Refer Slide Time: 31:16)



So, let me give you an example of how to eliminate such unreachable code, so this is the, you know piece of code which existed before it had a debugging instruction. So, if print equal to 1 goto L 1, otherwise goto L 2 and then L 1 had some debugging information print information it is printed out lot of information useful or debugging and L 2 of course was normal code. So, whenever the programmer wanted to debug he would set the variable print to 1 and then it would go and print the debugging information.

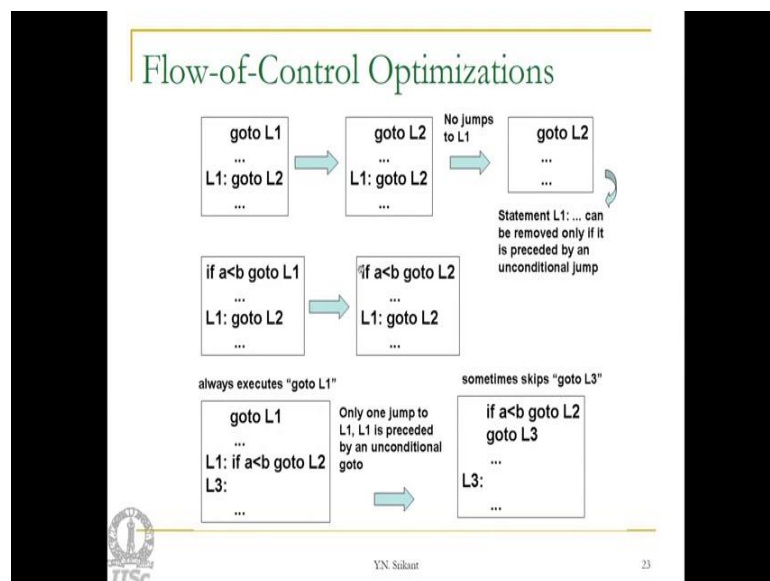
So, once the debugging was over the programmer did not do anything so the code remained as it is, but the print was set to 0. Then left at that point the first thing that the

compiler does in all these cases is you know instead of having this goto L 1 and then goto L 2 which a jump over the jump. So, this jumps over this jump it can change the condition as if print not equal to L 1 goto L 2, it did that automatically, now you know there is an improvement already you know.

So, instead of a check here and then a goto it can directly goto L 2 in case print is not required, since print was initialized to 0 by the programmer debugging phase is over. So, now this is production version the instruction, now the constant propagation took over print was actually replaced by 0. So, this became 0 not equal to 1 which was in turn found to be false by the compiler, sorry by 0 not equal to 1 is found to be true by the compiler.

So, this entire thing becomes a goto L 2, this part is not even needed it is evaluated permanently by the compiler itself. Now, we add a goto instruction here followed by print instructions we should not have any labels and then the normal piece of code. So, it is very clear that the print instructions which are here can never be reached by any part of the code again, so this has become unreachable code the compiler. Now, automatically removes such print instructions or any other instruction which exist between goto L 2 and L 2 because they are unreachable code. So, this is an example of how unreachable code can be removed in peephole optimization.

(Refer Slide Time: 34:06)



So, there are a couple of low of control optimizations which are very similar in spirit to the unreachable code removal for example, here is goto L 1 and then L 1 says goto L 2. So, we jump here and then again jump to L 2, so this can be obviously optimized as goto L 2 and the other one retained as goto L 2. So, we do not have to jump twice in order to goto L 2, now suppose you know we observe this code we never jump to L 1 and there are no other jumps to L 1 anywhere in the rest of the code let us say.

So, now we do not have to really keep this statement provided it can be you know it can be preceded it is preceded by an unconditional jump. So, otherwise it is possible that there is a jump to L 1 from some other place, so we cannot remove this piece of code without being assured just before this.

So, there is a, there are no jumps to L 1 and then all through from this part of the code to goto L 2 does not happen see one is a requirement is there are no jumps to L 1 from any other place. So, that makes sure that we do not come into L 1 from any other place, but what happens if this piece of code gets executed. Then we fall through and execute goto L 2 in order that this option is also taken care of there must be a, you know an unconditional goto just before L 1.

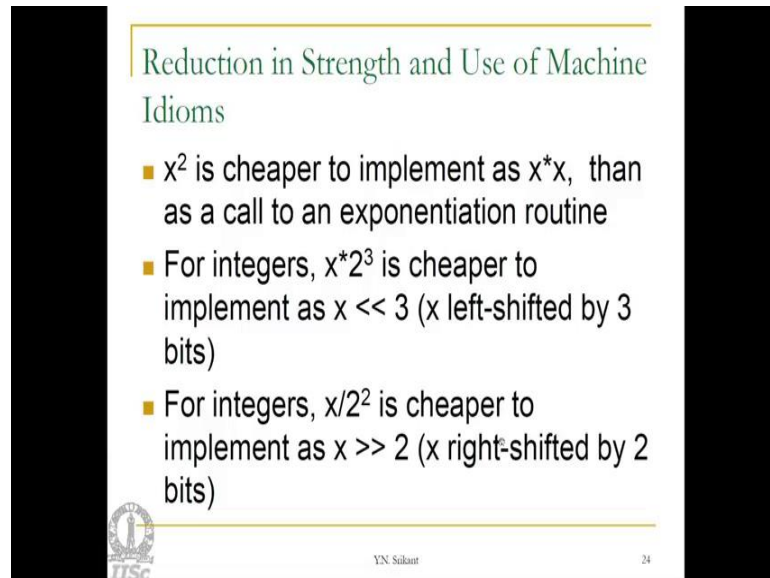
So, if that happens and this also happens we can remove the statement L 1 label, L 1, so this is the way in which we actually remove some of the unnecessary goto statements in the code. Then the next option next possibility is if a less than b then goto L 1 and L 1 has another goto L 2, so in such a case again we can replace it as if a less than b goto L 2 and keep the other one as goto L 2. So, in such a situation again we eliminate you know double jumps and replace it by a single jump instruction, so the other difficulty is there is a goto L 1 instruction and L 1 itself has if a less than b then goto L 2.

So, in such a case only one jump to L 1, you know in this case goto L 1 has all ways is always executed, so every execution goes to goto L 1. Then jumps to L 1, whereas only one jump to L 1, L 1 is preceded by an unconditional goto, so if that is the pattern that we detect then you know it is possible that we some we can sometimes skip the statement goto L 3, L 3.

So, if this pattern this code is now going to be replaced by this code, so a less than b goto L 2 and then goto L 3, so now in this case the code possibly skips goto L 3 sometimes

not necessarily always. Whereas, here we have always executed goto L 1, so the number of goto statements, which is executed by the program reduces by this transformation.

(Refer Slide Time: 37:43)



The slide is titled "Reduction in Strength and Use of Machine Idioms" and contains three bullet points. The first bullet point states that  $x^2$  is cheaper to implement as  $x*x$  than as a call to an exponentiation routine. The second bullet point states that for integers,  $x*2^3$  is cheaper to implement as  $x \ll 3$  (x left-shifted by 3 bits). The third bullet point states that for integers,  $x/2^2$  is cheaper to implement as  $x \gg 2$  (x right-shifted by 2 bits). The slide also features a logo for IITSC and the name YN Sikant in the footer.

Reduction in Strength and Use of Machine Idioms

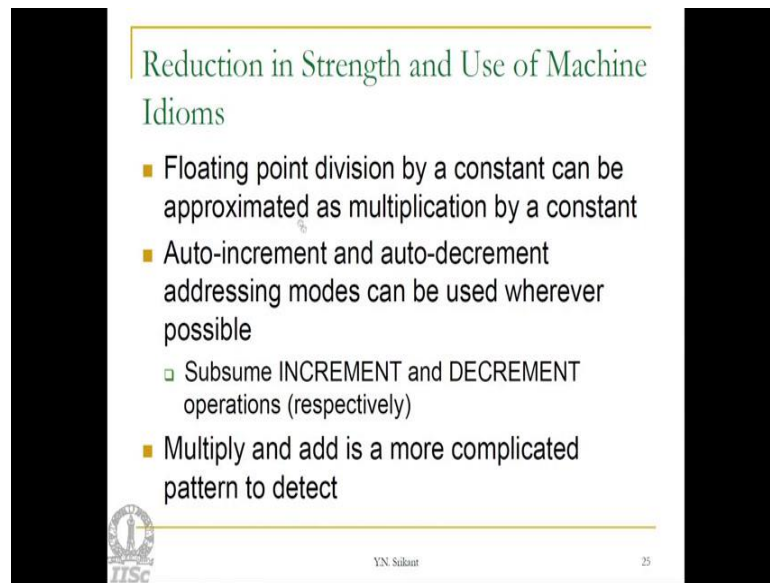
- $x^2$  is cheaper to implement as  $x*x$ , than as a call to an exponentiation routine
- For integers,  $x*2^3$  is cheaper to implement as  $x \ll 3$  (x left-shifted by 3 bits)
- For integers,  $x/2^2$  is cheaper to implement as  $x \gg 2$  (x right-shifted by 2 bits)

IITSC YN Sikant 24

Now, the next peephole optimization is called as reduction in strength and use of machine idioms, this is more like you know may sound like advice which is given to embedded system programmers as well. So, x square is obviously cheaper to implement as x star x and usually we never call a routine exponentiation routine to actually compute x square. But, in case it has been done and we find that there is a call to an exponentiation routine inserted by a naive programmer this can be replaced by you know x star x.

So, this is a very simple optimization which actually gives us much more improvement in execution speed. Similarly, x into 2 cube is obviously cheaper to be implemented as x left shift 3, that is x left shifted 3 times. So, you know every left shift operation actually is equivalent to multiplication by 2, so if we do it 3 times then it is multiplication 3 times by 2, so that is 2 cube. Similarly, every right shift operation is a division by 2, so x by 2 square can be implemented as a 2 right shift operations.

(Refer Slide Time: 39:18)



The slide is titled "Reduction in Strength and Use of Machine Idioms" and contains a bulleted list of optimization techniques. The list includes: floating point division by a constant approximated as multiplication by a constant; auto-increment and auto-decrement addressing modes used wherever possible, with a sub-bullet for subsuming INCREMENT and DECREMENT operations; and multiply and add as a more complicated pattern to detect. The slide also features a logo for IITSC and the name Y.N. Srikant.

- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect

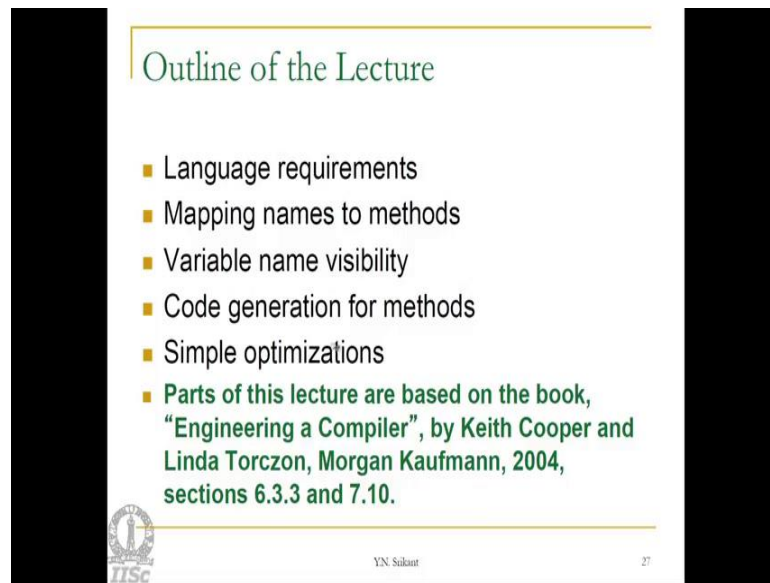
So, similar advice again floating point division is a by a constant you know can be actually the constant 1 by a constant can be evaluated by the compiler and that division can now be replaced as a multiplication by a constant. So, this can be done by a peephole optimizer then if the machine has auto increment and auto decrement addressing modes there can be used wherever possible. But, how does one use it you know it is possible to subsume increment and decrement operations respectively into these addressing modes.

So, whenever we have these addressing modes available we can and there is an increment operation which can be combined with it. So, it can be subsumed into this addressing mode and appropriate instructions can be emitted by the peephole optimizer there are more complicated patterns which can be detected such as multiply and add.

Now, this is a much more complicated pattern and not many peephole optimizers actually checked and complete it, so from now we move on to the, you know this is the code generation that we have been discussing so far. Now, let us see how to generate code for object oriented programming languages so far we discussed code generation for C type of programming languages. Now, let us see how to generate code for object oriented languages such as java and C plus plus.




(Refer Slide Time: 40:57)



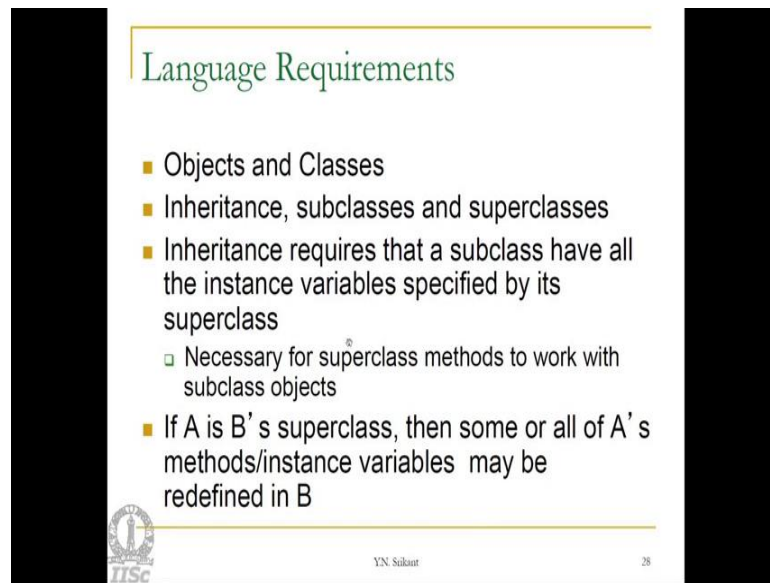
Outline of the Lecture

- Language requirements
- Mapping names to methods
- Variable name visibility
- Code generation for methods
- Simple optimizations
- Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.

 YN Srikant 27

So, now let us first overview the requirements of the language and then you know the rules that are used or naming the mapping the names to methods etcetera. So, the variable names and their visibility the rules or these also have to be studied then we come to the code generation methods. So, some of the simple optimizations which can be performed for such languages are also going to be reviewed here and some parts of this lecture are, you know based on an excellent chapter in the book engineering a compiler by Cooper and Torczon.

(Refer Slide Time: 41:50)



The slide is titled "Language Requirements" in a green serif font. It contains a bulleted list of requirements. The first bullet is "Objects and Classes". The second is "Inheritance, subclasses and superclasses". The third is "Inheritance requires that a subclass have all the instance variables specified by its superclass", which has a sub-bullet: "Necessary for superclass methods to work with subclass objects". The fourth is "If A is B's superclass, then some or all of A's methods/instance variables may be redefined in B". At the bottom left is the IITSC logo, and at the bottom center is the text "Y.N. Sankar" and the number "28".

- Objects and Classes
- Inheritance, subclasses and superclasses
- Inheritance requires that a subclass have all the instance variables specified by its superclass
  - Necessary for superclass methods to work with subclass objects
- If A is B's superclass, then some or all of A's methods/instance variables may be redefined in B

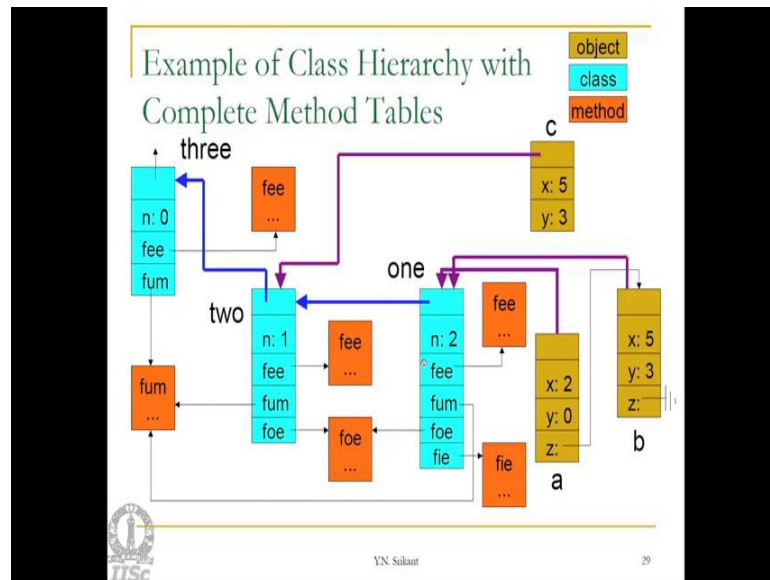
So, let us review the language requirements to make a language as object oriented first of all the language which is set to be object oriented must have you know a declaration for objects and it must have a declaration or classes. So, C plus plus and java both satisfy this requirement, a second important requirement is that there must be inheritance. So, you should be able to declare sub classes and have available super classes this is obviously satisfied.

So, inheritance is property by which we can say here is a class a, now I declare another class b which has inherited all the properties from the class a. So, that is how inheritance works we see example of this very soon, so inheritance requires that a subclass have all the instance variable specified by its super class. So, in other words whenever we instantiate a you know a class and we produce an object then you know we obviously are going to have variables of that particular class instantiated in the object that is possible.

But, here we are talking about a different variety all together you have a class then we produce a subclass using inheritance. So, the subclass will have you know all the variables instance variables specified by the super class itself, so this is something different from what the objects specify. So, whatever the super class has will all be available to the subclass, this is a kind of necessary, the reason is the super class will have a lot of methods which work with its instance variables.

Now, when we declare a subclass we may have methods of the super class working on the objects of type subclass itself. So, when we want to this to happen unless the rule is satisfied there would be difficulties in execution then if a is b's super class then some are all of a's you know a methods and instance variables may be redefined in b.

(Refer Slide Time: 44:26)



So, let us look at an example to understand all these concepts in a better way the cover code is very simple, so this brown indicates objects this greenish blue indicates classes and red indicates methods. So, there are three classes here this is the highest level super class that is called as 3, there is a super class called 2 which is a subclass of this class 3. So, this is class 2 which is a subclass of class 3, but 2 itself is a super class as far as class one is concerned. So, the classic hierarchy is three is at the highest level 2 is the next level and 1 is at the lowest level the implication of this is whatever methods.

Here, variables that 3 declares will all be available in 2 and whatever is declared in 3 and 2 will all be available in 1 as well. So, it does not mean these two classes cannot define something on their own they can add extra you know, for example let us go through the list to understand this better. So, there is a static variable n which is available in all of them there is a, you know a method called fee which is actually defined in 3, but it is redefined in 2 and 1. So, each of them have their own fee, see this is fee of 2 and this is fee of 1, so whatever is available here will not be used by 1 and 2 they will be redefined.

Then there is a method called `fum` which is defined by class 3 `fum` is actually shared by both class 2 class and class 3, so this is the difference. So, `fee` is private to not private, sorry `fee` is not shared between 1 to 3, it is available to 3 it is redefined into and it is redefined in one as well. Whereas, the method `fum` is available to both three two and one and they are also shared, so this is they are also sharing the method form that is very important.

Now, `4` is not available in the class 3 at all it is a new method which is defined in class 2 it was available in 3 and, now this is shared between class 1 and class 2. So, finally this method `fie`, `fie` is available within class 1 alone it is not defined in 2, it is not defined in 3 it is available only in you know class 1.

So, the whatever is available in the super class may be redefined in the sub class it can share the available one with the super class or and it can also introduce new methods of its own in the subclass itself. So, apart from this it is also possible to introduce extra variables, for example if we see `c`, for example is a variable which is an instance of class number class 2. So, it has a 2 you know variables within it `x` and `y` 2 fields, whereas the variables `a` and `b` are instances of the class 1, so these are 2 objects which are instances of class 1. So, `x` and `y` of course are available in both `a` and `b` simply because they were available to all instances of class 2 which is the super class of class 1. But, `z` is extra it is available only to instances of class one and it is not available to any instances of class 2. So, these are some of the general difficulties and specialties which are introduced in the language because of inheritance.

(Refer Slide Time: 49:04)

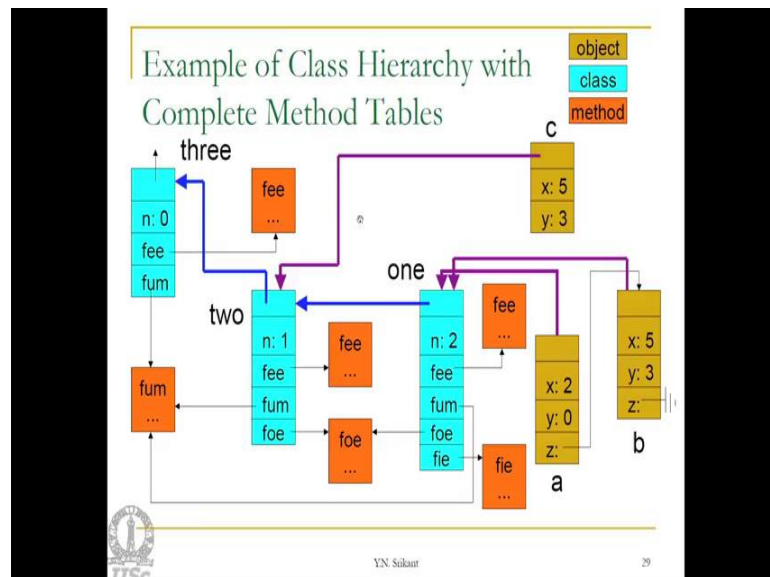
### Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee()* invokes *one.fee()*, *a.foe()* invokes *two.foe()*, and *a.fum()* invokes *three.fum()*
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

ITSC YN Sikaur 30

So, now let us look at a few more details method invocations are not always static calls what is a static call we know the address of the method which is to be called that is when it is static i it is not static. Then there are special things that need to be done in order to call that particular method, so for example a dot fee is a method call it invokes 1 dot fee.

(Refer Slide Time: 49:40)




So, let us look at it we have you know object a here, so it is actually an instance of class 1 and we are calling a dot fee, so that is the method of class 1 itself.

(Refer Slide Time: 49:58)

### Mapping Names to Methods

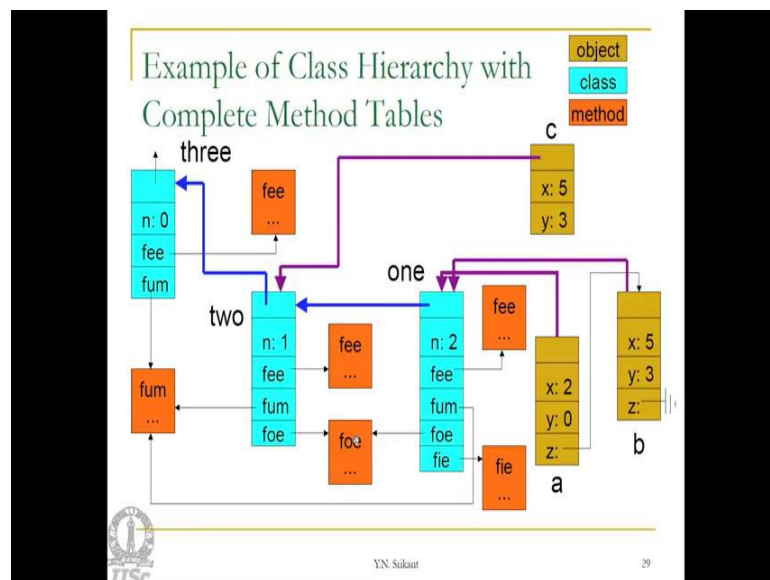
- Method invocations are not always static calls
- *a.fee()* invokes *one.fee()*, *a.foe()* invokes *two.foe()*, and *a.fum()* invokes *three.fum()*
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)



YN Sikaur 30

So, that is what we want here 1 dot fee, whereas if we call the function method a dot 4 it invokes 2 dot 4.

(Refer Slide Time: 50:10)



So, again going back to this, so if we say a dot 4 the method our is not defined by class one but, it is defined by its super class 2. So, it is not available within class 1, but it has to go and get it from class 2, so that is the difference, so even though class this object a belongs to class 1. So, the method call actually now invoked you know supposed to be which is invoked does not belong to class 1, but it is invoked from class 2 even worse.

(Refer Slide Time: 50:46)

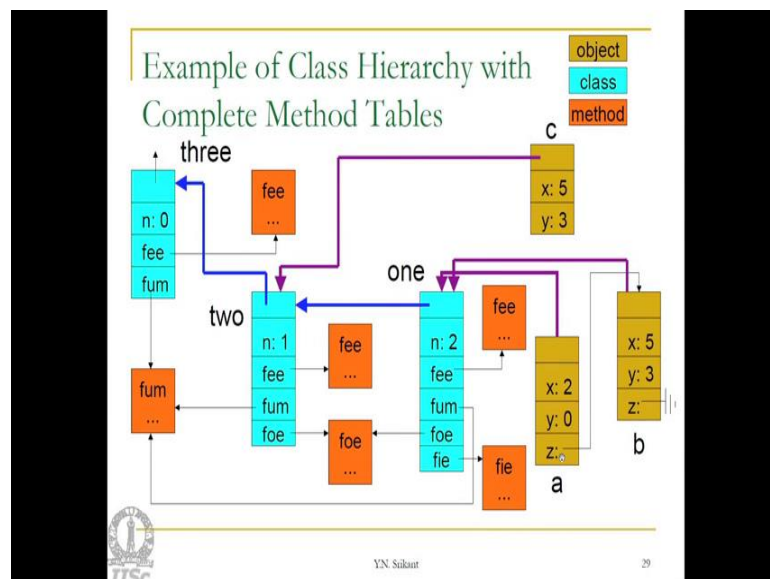
### Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee()* invokes *one.fee()*, *a.foe()* invokes *two.foe()*, and *a.fum()* invokes *three.fum()*
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

ITSC YN Sankar 30

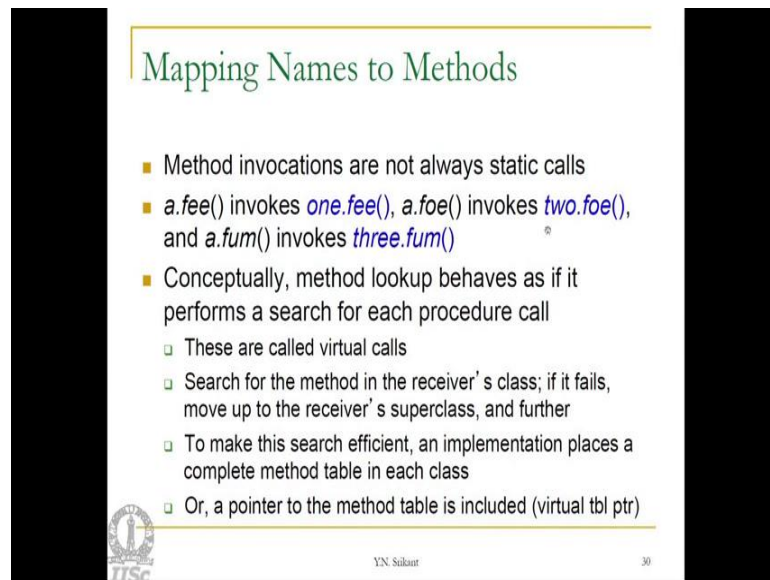
So, suppose we call a dot fum.

(Refer Slide Time: 50:52)



So, a dot fum even though a is an object of class 1 it is a fum is not defined either in class 1 or in class 2 it is actually defined in class 3 which is 2 classes away it is a super class of both one and 2. So, after having searched one it goes to class 2, it is not found there either it will have to check class three then get hold of the code for fum and then execute it.

(Refer Slide Time: 51:22)



The slide is titled "Mapping Names to Methods" and contains the following content:

- Method invocations are not always static calls
- *a.fee()* invokes *one.fee()*, *a.foe()* invokes *two.foe()*, and *a.fum()* invokes *three.fum()*
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

At the bottom left of the slide is the IITSC logo, and at the bottom center is the text "YN Sankar" and the number "30".

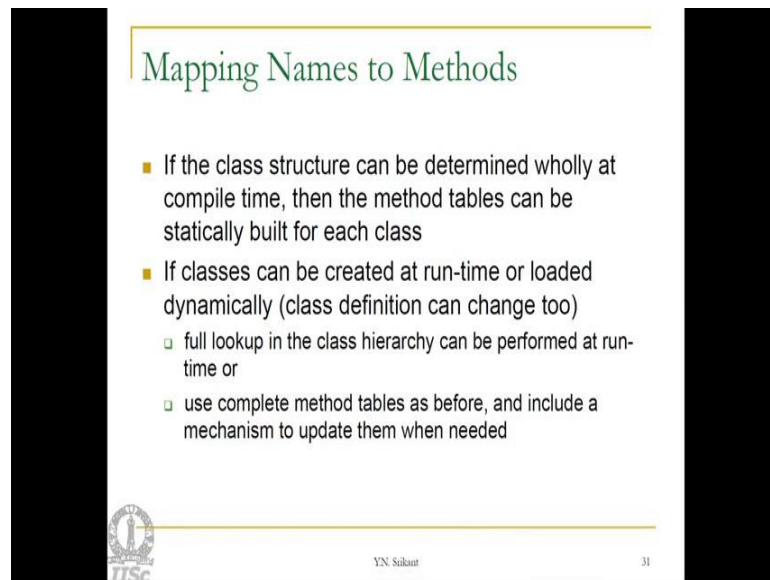
So, this is the, this is what I mean by saying it is not always the static call it is not even known whether the code corresponding to the method exists within that particular class from which we are making a call or it belongs to the super call. So, that is not even known then how does this method look up really work in practice. So, method look up behaves as if it is making a search for a each procedure call, so these are called as virtual calls because they are not static calls they must be given a name.

So, they are really called virtual calls, so the first search is always within the first the parent class, so the method the object actually makes a call to a method, the method we searched in the parents. So, you know method table if it is not found there the method tables of the super class of the parent are searched and the search goes upwards to the various super classes available in the hierarchy.

Now, how do you make this search very efficient that the question one possibility is we copy the entire method table from each of these super classes into the parent class itself. So, each one of the classes will contain a complete method table, so that the implementation becomes very efficient. Now, the second possibility is instead of including the entire method table, we could introduce actually a pointer to the method table and that is called as a virtual table pointer.



(Refer Slide Time: 53:18)



The slide is titled "Mapping Names to Methods" in a green serif font. It contains a bulleted list of three main points, with the second point having two sub-points. The footer includes a circular logo on the left, the text "Y.N. Sikant" in the center, and the number "31" on the right.

- If the class structure can be determined wholly at compile time, then the method tables can be statically built for each class
- If classes can be created at run-time or loaded dynamically (class definition can change too)
  - full lookup in the class hierarchy can be performed at run-time or
  - use complete method tables as before, and include a mechanism to update them when needed

The next thing is to study the mapping of names to methods, so if the class hierarchy and the class structure can be determined completely at you know compile time. Then we can actually build the entire method table at compile time and put it into each class that is very easy right, so we know all the classes you know all the methods in every one of the classes. So, we did the class hierarchy then we determined the various method you know build all the method tables place the entire method table into each class to that is fairly straight forward.

But, there are situations in languages such as java where the class does not exist at compile time the class may be produces dynamically during execution. But, it can be possibly loaded dynamically via internet when the program execution after the program execution begins. So, we suddenly call let us say a method which of at last which does not exist, now then the method will be downloaded via internet rather not the method the class will be downloaded via the internet. Then you know we create the class in the data structures of the class will be created we create an object of that particular class and we may have to execute the method corresponding to that particular class later on.

So, such dynamic execution, you know rather dynamic loading of classes introduces difficulties into our mapping names to methods and etcetera. So, a full lookup in the class hierarchy can be performed at run time to take care of this, so we build the class hierarchy at run time. Then we performed the lookup of course we use complete method

tables as before, but we need to update the method tables whenever there is a change in the class hierarchy or a new class is added and so on so forth. We will stop our discussion at this time and continue in the next class.

Thank you.