

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

Module - 6
Lecture - 23
Run-time environments Part-4
Control-flow graph and local optimizations Part-1

(Refer Slide Time: 00:22)

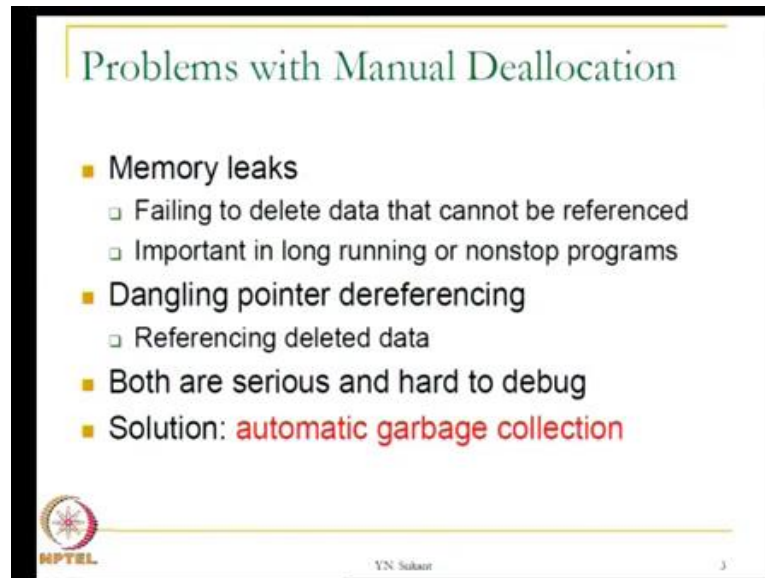
Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation (in part 2)
- Activation records (in part 2)
- Static scope and dynamic scope (in part 3)
- Passing functions as parameters (in part 3)
- Heap memory management (in part 3)
- Garbage Collection

NPTEL
YN Srikant

Welcome to part four of the lecture on run time environments. Today we will continue with garbage collection and you know see how exactly two types of garbage collectors work, and so on.

(Refer Slide Time: 00:34)



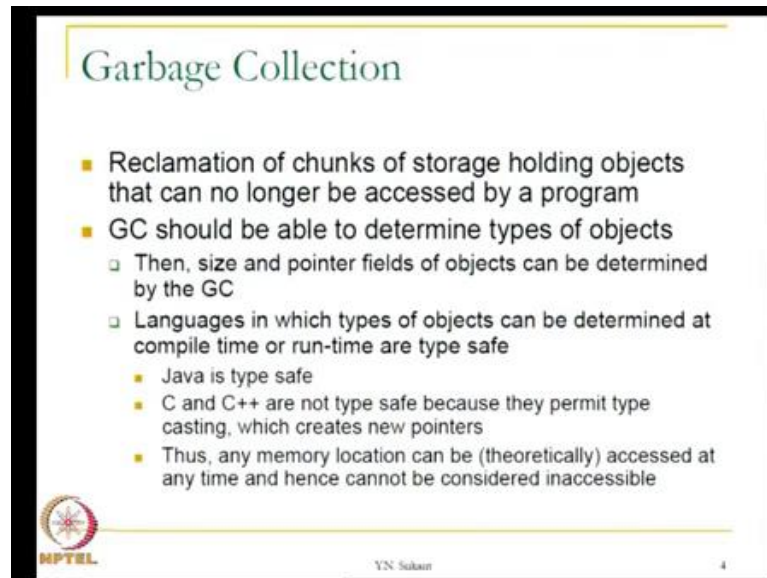
The slide is titled "Problems with Manual Deallocation" in a green serif font. It contains a bulleted list of four items. The first item is "Memory leaks", which has two sub-points: "Failing to delete data that cannot be referenced" and "Important in long running or nonstop programs". The second item is "Dangling pointer dereferencing", with a sub-point: "Referencing deleted data". The third item is "Both are serious and hard to debug". The fourth item is "Solution: automatic garbage collection", where "automatic garbage collection" is written in red. In the bottom left corner, there is a circular logo with a star and the text "NPTEL". In the bottom center, it says "Y.N. Srikant" and in the bottom right corner, it says "3".

- Memory leaks
 - Failing to delete data that cannot be referenced
 - Important in long running or nonstop programs
- Dangling pointer dereferencing
 - Referencing deleted data
- Both are serious and hard to debug
- Solution: automatic garbage collection

So, to do a bit of recap let us see what exactly are the problems with manual deallocation, that is using free etcetera. So, there can be memory leaks which are not detected in the program it is very hard to debug such memory leaks. So, the memory leak is you know failing to delete data that cannot be referenced. So, basically we have forgotten to delete the nodes which we are not using anymore in a tree or in a graph, etcetera.

And then there is a second problem of dangling pointer dereferencing, so we have forgotten to you know remove the references to deleted data. So, this could be you know a program error, which is very hard to debug as well the solution at least partial solution to these problems is automatic garbage collection.

(Refer Slide Time: 01:33)



Garbage Collection

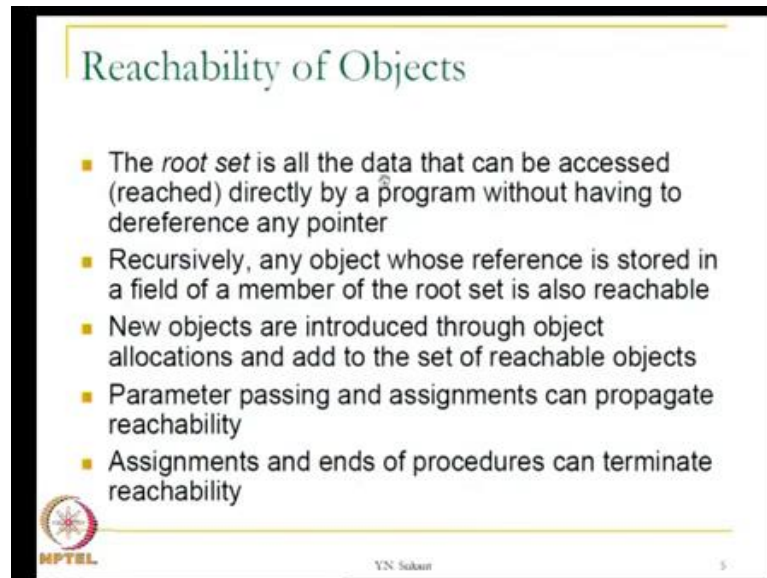
- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
 - Then, size and pointer fields of objects can be determined by the GC
 - Languages in which types of objects can be determined at compile time or run-time are type safe
 - Java is type safe
 - C and C++ are not type safe because they permit type casting, which creates new pointers
 - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

NPTEL VN Srikar 4

So, garbage collection is nothing but reclamation of storage and the storage is not being used by the program, so we want to free such locations and use them again. It is possible to you know do this provided the types of objects are known to the garbage collector. The reason why we require the types of objects is that the sizes of these objects cannot be determine without knowing the type.

Secondly, it is possible that the pointers, which are embedded inside the objects you know also refer to some you know runtime object. And it is not possible to determine whether there are pointers inside an object without knowing the type of that particular object. So, java for example, is you know is type safe because you can determine the types of objects either at compile time or at runtime c and c plus plus are not.

(Refer Slide Time: 02:41)



Reachability of Objects

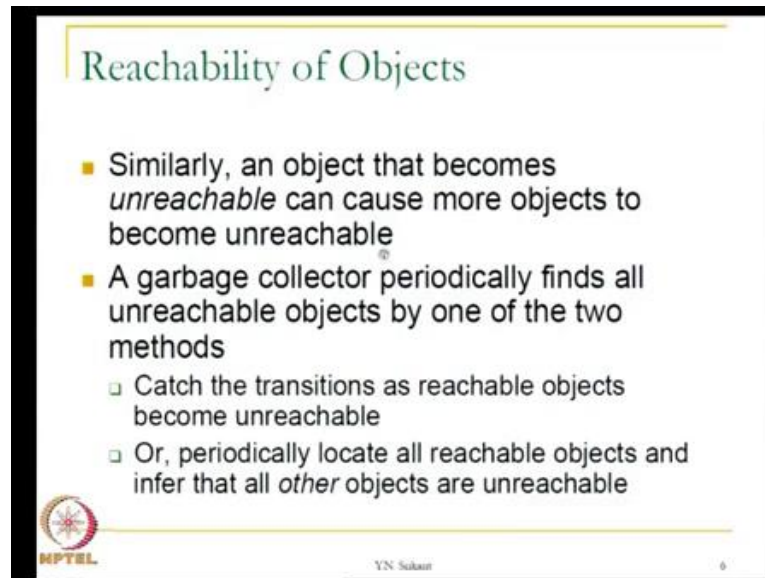
- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

NPTEL Y.N. Sankar 5

So, the basic mechanism by which we claim garbage is using the concept of reachability of objects. So, what we really want is to determine the root set which is nothing but the set of all data that can be accessed directly by a program without having to dereference any pointer. That is if you consider the pointer variables which have been declared by the programmer at the highest level in the program.

We are looking at these pointers as a pointer variables as the root set. So, from that point you know from that set we reach the objects you know using one level of dereferencing. And we also look at the objects which can be reached from the at the second level and so on and so forth. So, let us see how this reach ability can be used in garbage collection.

(Refer Slide Time: 03:48)



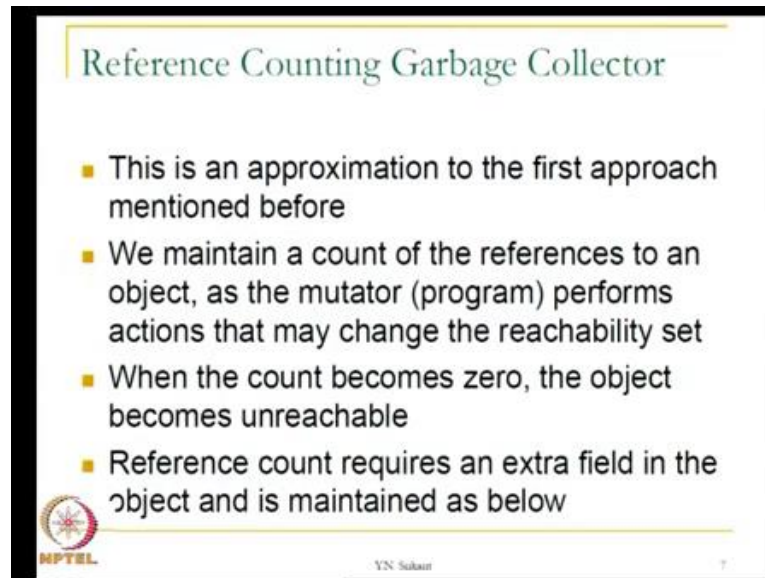
The slide is titled "Reachability of Objects" in a green font. It contains a bulleted list of three main points. The first point is a yellow square followed by the text "Similarly, an object that becomes *unreachable* can cause more objects to become unreachable". The second point is a yellow square followed by "A garbage collector periodically finds all unreachable objects by one of the two methods". This second point has two sub-points, each marked with a small square: "Catch the transitions as reachable objects become unreachable" and "Or, periodically locate all reachable objects and infer that all *other* objects are unreachable". In the bottom left corner, there is a circular logo with a star and the text "MPTEL". In the bottom center, it says "YN Sakari". In the bottom right corner, there is a small number "6".

- Similarly, an object that becomes *unreachable* can cause more objects to become unreachable
- A garbage collector periodically finds all unreachable objects by one of the two methods
 - Catch the transitions as reachable objects become unreachable
 - Or, periodically locate all reachable objects and infer that all *other* objects are unreachable

So, that is also possible I forgot to mention that parameter passing Assignments can propagate reach ability. And of course, assignments and ends of procedures can terminate reach ability of these objects. Similarly, an object that becomes unreachable can cause more objects to become reachable unreachable. So, that is you know if you have a pointer you can reach an object through it. And if that object has more pointers in it you can reach more objects through it etcetera. Similarly, if you actually free an object, which was being you know reached by a pointer.

Now, the objects which were reached from the pointers within the object also become unreachable through this pointer. So, this is what we mean by the propagation of unreachability. The garbage collector you know periodically finds all the unreachable objects using one of the two methods. The first one says we know that the objects change state. So, when the objects become unreachable at that transition from reachable to unreachable we catch the, you know object and say now let me free it. So, second is periodically we scan the entire heap of the program and then locate the reachable objects and whatever is left over is the unreachable set.

(Refer Slide Time: 05:25)



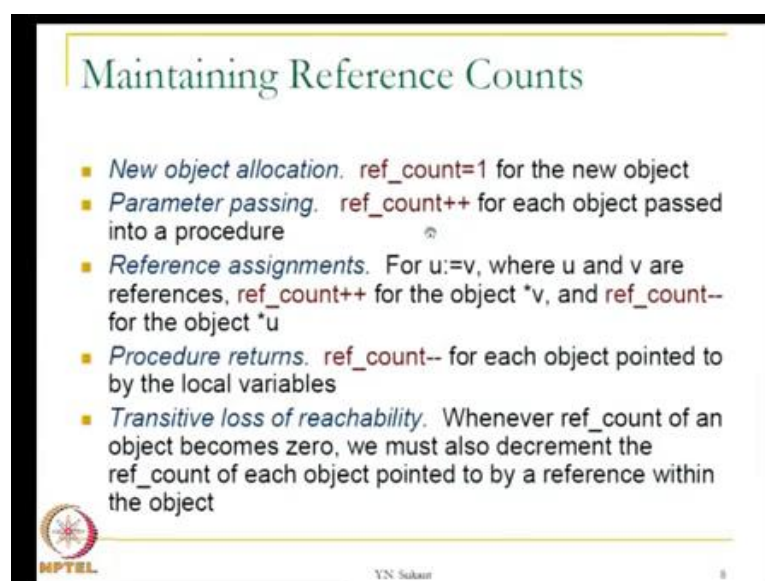
Reference Counting Garbage Collector

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

NPTEL YN Sakain 7

The first method that is the reference counting garbage collector is an approximation to the approach that I you know, the first approach that is we catch the objects which become unreachable at the time of transition. So, how do we do this basically we maintain a count of the references to an object so. And when the count becomes 0 the object becomes unreachable, but how do we maintain the count. So, whenever you know the program perform certain actions and that changes the reach ability of that object. We modify the count as well, I will give you more details of this very soon and the reference count requires an extra field in the object. So, we maintain it you know using these rules.

(Refer Slide Time: 06:22)



Maintaining Reference Counts

- *New object allocation.* `ref_count=1` for the new object
- *Parameter passing.* `ref_count++` for each object passed into a procedure
- *Reference assignments.* For `u:=v`, where `u` and `v` are references, `ref_count++` for the object `*v`, and `ref_count--` for the object `*u`
- *Procedure returns.* `ref_count--` for each object pointed to by the local variables
- *Transitive loss of reachability.* Whenever `ref_count` of an object becomes zero, we must also decrement the `ref_count` of each object pointed to by a reference within the object

NPTEL YN Sakain 8

Now, the rules for maintaining reference counts there is a let us say there is a new object allocation. So, that means the object comes into you know is born right so using new or any other mechanism. So, we set the reference count as 1 for this new object so that is quite fair because we have created a new object and that is being pointed to by a pointer. So, the reference count is one at that time.

Parameter passing so, let us say objects are being passed into a procedure as parameters. So, that means from now on the parameter also refers to the object apart from the original number of references to that object. So, it is only fair that we do an increment operation for each object on the reference count counter, for each object passed into a procedure. So, this is parameter passing and its effect on the reference count, then there are reference assignments. So, that is u and v are references so and we are saying u equal to v this is an assignment.

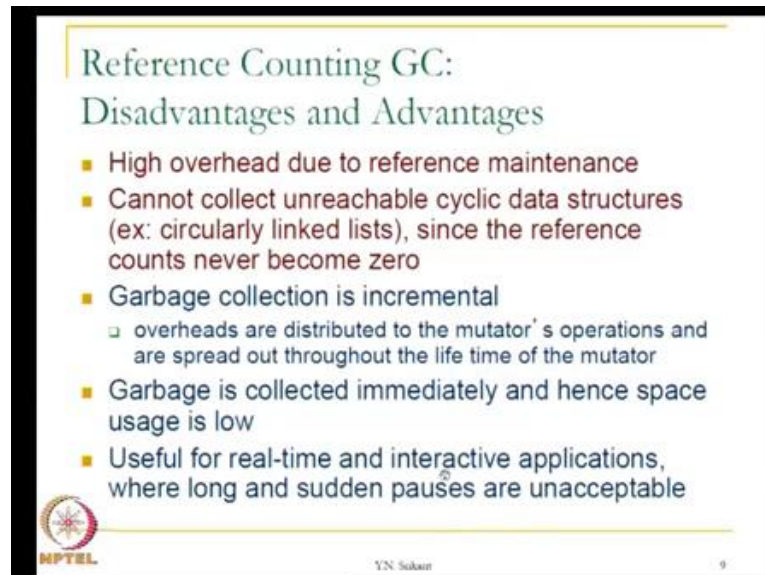
So, u is pointing to some object v is also pointing to some other object and from now on u will point to the object that v is pointing to so, this is a pointer copy operation. So, obviously for the object which is pointed to by v that is $\text{star } v$. The reference count has to be incremented by 1 because now from now on u will also point to it. And now, u has a kind of c 's to point to the old object so, reference count is reduced for the object which is pointed to by u that is $\text{star } u$.

So, this is how the reference assignment propagates you know reach ability and then and then of course, it ends the reach ability for the objects pointed to by u as well. What about procedure returns so, we know that the local variables die once a procedure returns. So, we do a reference count minus minus that is decrement operation on the reference counts, for each objects which is pointed to by the local variables, local pointers. Then there is transitive loss of reach ability as well so whenever an object have gets its reference count to 0.

We must now trace further we must locate the reference count of each object pointed to you know by a reference within the object. So, we have an object there are more pointers within it, those pointers will also be pointing to some objects. So, if an object gets its reference count to 0 then you know you must decrement the reference count of the objects pointed to by the pointers inside as well. So, this must be done transitively and it


will go on until no more decrement operation can be performed in any reference count of any particular object.

(Refer Slide Time: 09:38)



Reference Counting GC:
Disadvantages and Advantages

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
 - overheads are distributed to the mutator's operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

 NPTEL

Y.N. Subram 9

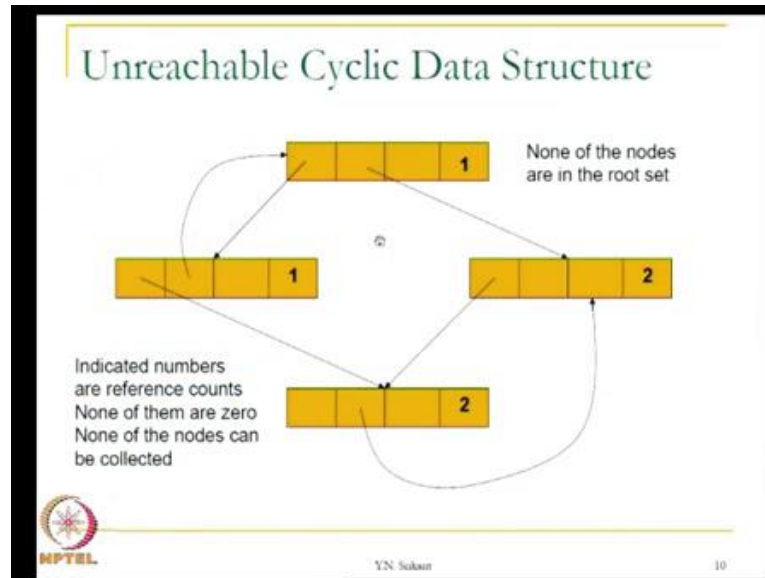
So, this is the reference counting you know method of garbage collection. So, basically we whenever we find that the reference count becomes 0, we return that object to the heap so garbage collection is in some sense incremental. So, overheads are distributed to the mutator operations and are spread out throughout the lifetime of the mutator. So, we do not have you know a huge amount of time, which is spent in garbage collection at a single point in time, but it is distributed throughout the you know programs operations.

So, garbage is collected immediately and hence space usage is low and usage is also very efficient, we are not delaying you know the collection of garbage. So, for real time systems and interactive applications, such garbage collection is very useful because long and sudden pauses are unacceptable in such an environment. So, if we take away too much time during the applications run to perform garbage collection, it would obviously introduce long and sudden pauses.

So, to avoid this an incremental garbage collection using reference counting is a very useful mechanism, but then these are the advantages. So, then there were also a few disadvantages there is a very high overhead due to reference maintenance because every time the reference changes, we know that the reference counter has to be either incremented or decremented. And, there is also a more serious problem the reference

counting garbage collector cannot collect unreachable cyclic data structure. Such as, circularly linked lists I will show you an example of this very soon. Since, the reference counts never really become zero this is something very serious.

(Refer Slide Time: 11:50)



So, take this data structure so each of these is a 4 field object so, this pointer inside this object points to this node and this points to this node. So, here is a cycle of references, this pointer points to this node and then this pointer points to this node and again this pointer points to this node. So, here is another cycle and the second pointer here also points to this node. So, this object has a reference count of 1 because of this pointer pointing to it.

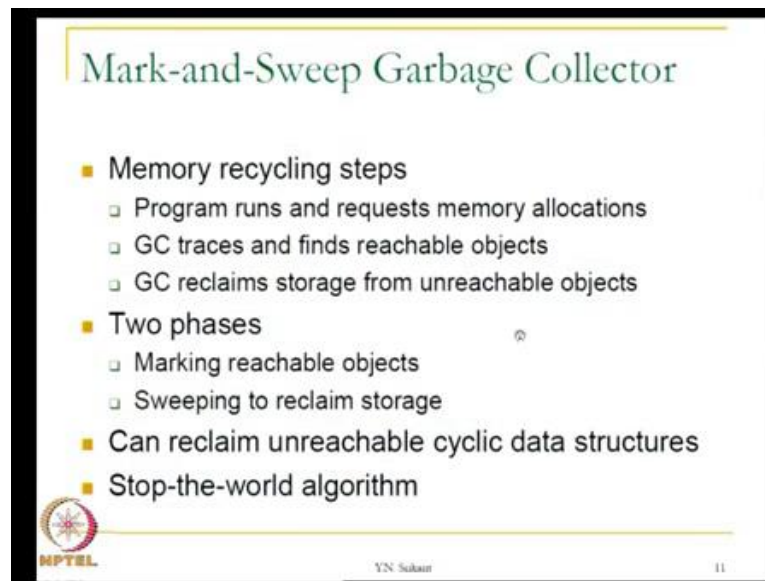
This object has a reference count of 1 again because of this pointer pointing to it. This object has reference count 2 because there are 2 objects, 2 pointers pointing to it. And then this object also has a reference count of 2 because there are 2 objects pointing to it this and this. So, what it so happens that none of the objects here can be reached through any other useful program pointer variable or any other variable, any other pointer inside the program.

So, this is basically garbage none of these nodes can be reached from anywhere in the program, but unfortunately the reference counts have not become 0. Therefore, none of these nodes will be returned to the storage pool the heap. So, this is a serious issue none

of these nodes are in the root set either or transitively even in the root set. So, this storage will be permanently claimed and you know it remains in the program.

So, if such circular data structures you know keep accumulating then even though they are all useless it will not be possible to return them to the storage pool. So, this is a big disadvantage as far as the reference counting garbage collector is concerned. So, how do we get rid of this problem?

(Refer Slide Time: 14:03)



The slide is titled "Mark-and-Sweep Garbage Collector" in a green serif font. It contains a bulleted list of features and steps. The first bullet is "Memory recycling steps" with three sub-bullets: "Program runs and requests memory allocations", "GC traces and finds reachable objects", and "GC reclaims storage from unreachable objects". The second bullet is "Two phases" with two sub-bullets: "Marking reachable objects" and "Sweeping to reclaim storage". The third bullet is "Can reclaim unreachable cyclic data structures" and the fourth is "Stop-the-world algorithm". In the bottom left corner, there is a circular logo with a star and the text "NPTEL". In the bottom center, it says "Y.N. Srikant" and in the bottom right corner, it says "11".

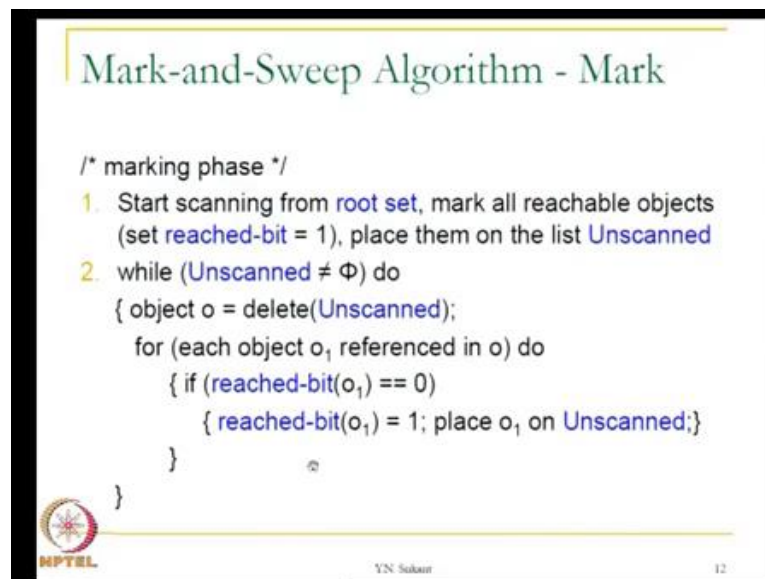
- Memory recycling steps
 - Program runs and requests memory allocations
 - GC traces and finds reachable objects
 - GC reclaims storage from unreachable objects
- Two phases
 - Marking reachable objects
 - Sweeping to reclaim storage
- Can reclaim unreachable cyclic data structures
- Stop-the-world algorithm

So, let us look at a different type of garbage collector, mark and sweep garbage collector. So, the briefly again the memory recycling steps in such a garbage collector are the program runs it request memory locations. The garbage collector traces and finds reachable objects then the garbage collector reclaims storage from unreachable objects. So, the entire heap memory is scanned by the garbage collector in both these passes finding reachable objects and claiming unreachable objects.

So, as the name indicates there is a marking phase to mark the reachable objects and there is a sweeping phase to reclaim the unreachable objects and return them to the storage pool. The advantage of this is it can claim unreachable data structures such as, circularly link lists and so on. And the other feature is the nature of the algorithm it is a stop-the-world algorithm. In other words, when the program runs out of memory the program you know calls the garbage collector, when the when an allocation is made the allocation fails.

So, the garbage collector is called to perform garbage collection, the garbage collector does these 2 operations of marking and sweeping this may. And during this time the rest of the program does not run so this is called as a stop-the-world algorithm. And this is the one which may introduce pauses in the program and it hurts real time and interactive programs.

(Refer Slide Time: 15:54)



```
Mark-and-Sweep Algorithm - Mark

/* marking phase */
1. Start scanning from root set, mark all reachable objects
   (set reached-bit = 1), place them on the list Unscanned
2. while (Unscanned ≠ Φ) do
   { object o = delete(Unscanned);
     for (each object o1 referenced in o) do
       { if (reached-bit(o1) == 0)
         { reached-bit(o1) = 1; place o1 on Unscanned;}
       }
   }
```

So, let us look at the details of the algorithm, the algorithm basically is quite simple there are many variations of the mark and sweep algorithm. We are going to look at one variety the simplest one and the rest can be read from the textbooks. So, what is the marking phase start scanning from the root set, first step in the marking phase is start scanning from the root set. Mark all the reachable objects that is set reachable be reached bit equal to 1 and place them on the list called unscanned.

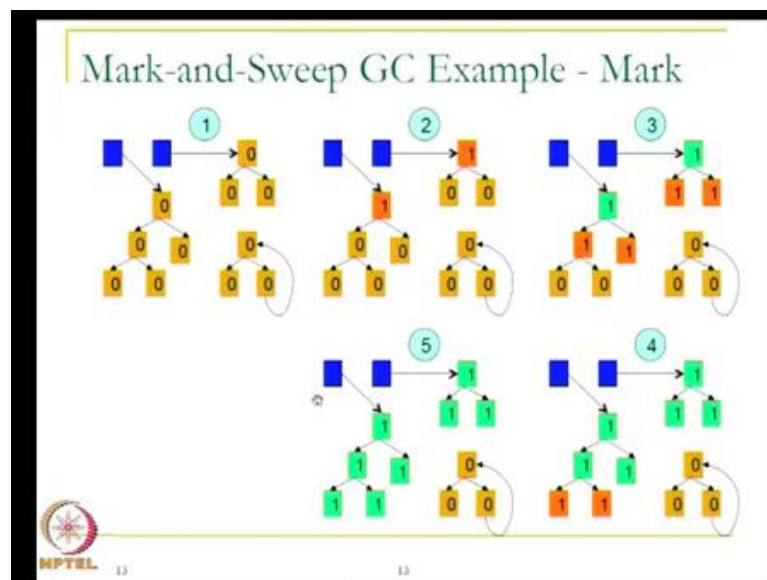
So, there your root set is as usual the set of pointer variables which are declared by the programmer. So, all these all the objects which are reached from the root set will be marked as, marked with reached bit equal to one and they will be placed on a list called unscanned. Now, we must do the marking you know rather a progressively so there is a loop which continues until unscanned does not contain any object so, while unscanned not equal to phi do.

So, take out an object from the unscanned list so object O equal to delete unscanned. And then for each object O 1 referenced in O so there are pointers within the object so

each one of these pointers is now actually traced. So, if the object is already you know is not yet reached so reached bit 0 1 equal to 0. Then we say the we make it 1 reached bit 0 1 equal to 1 and place 0 1 on unscanned so this goes on and on.

So, once all the pointers in the object O you know object O are completed we take the next object from the unscanned list and continue. So, in this process every one of the objects which actually can be reached, will be reached and each of these objects will have their reached bit set to 1. All the you know objects which have reached bit equal to 0 will be unreachable and they can be claimed during the next phase.

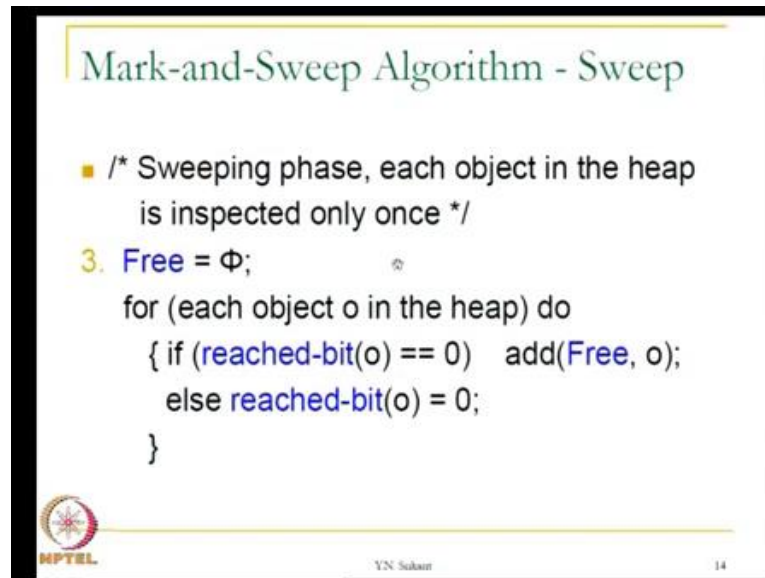
(Refer Slide Time: 18:21)



So, here is an example to show how mark works so this is the root set the blue variety. So, these are the two variables programmer variables which used to access these data structures. And this is the storage which cannot be reached it is garbage and it needs to be collected. So, we start from the root set and then mark the objects these two objects as 1 and then from these two objects we progressively go to these and these and so on.

So, we mark these two as 1 and these two again as 1. Then in the next step we go to these two objects and mark them as 1. So, we now have all the reachable objects marked as 1 and the unreachable object remains with reached bit equal to 0, the unscanned list now is empty so the marking phase ends.

(Refer Slide Time: 19:23)

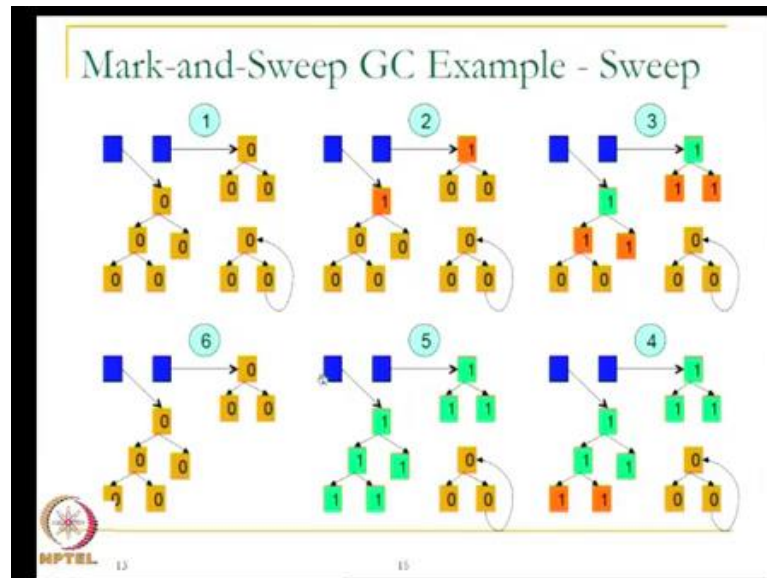


The slide is titled "Mark-and-Sweep Algorithm - Sweep". It contains a code snippet for the sweeping phase. The code starts with a comment: "/* Sweeping phase, each object in the heap is inspected only once */". This is followed by a line "3. Free = Φ ;" where Φ is the Greek letter phi. Then, there is a loop: "for (each object o in the heap) do". Inside the loop, there is an if-statement: "{ if (reached-bit(o) == 0) add(Free, o); else reached-bit(o) = 0; }". At the bottom left of the slide is the NPTEL logo, and at the bottom right is the number "14".

The next phase is the sweep phase so again the entire heap is scanned here. So, sweep phase each object in the heap is inspected exactly once not more than once, we will see why it is incorrect to look at it more than once. So, to begin with the free list is phi we had nothing on it and we look at every object in the heap. And if the reached bit of that object is 0 we just dispose it off, add it to the free list. If the reached bit was 1, then that means, the object was useful so we set it to 0.

So, you see for the objects which had their reached bit equal to 1 we have now reset it to 0. So, looking at the object again would be incorrect because it would you know send all these objects to the free space. These were really useful objects so we should not be looking at them again. So, let us see how it works in this case.

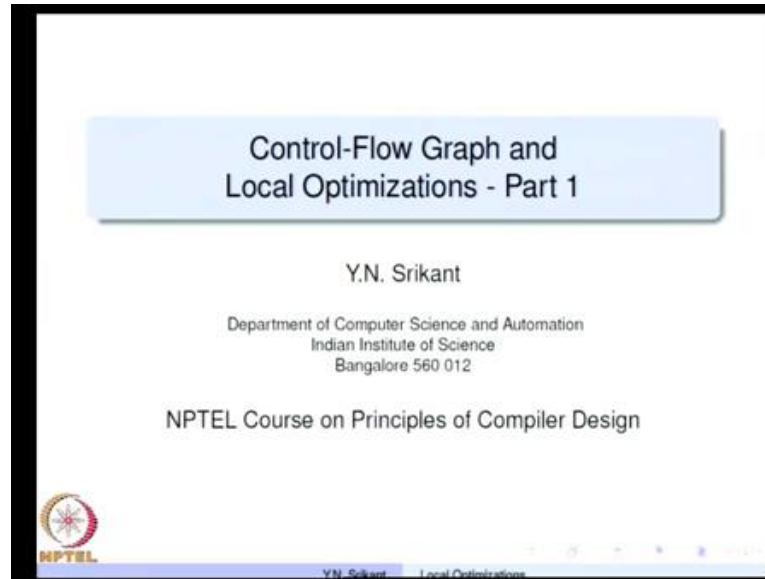
(Refer Slide Time: 20:28)



So, up to this was the marking phase so all the reachable objects have been marked, these are the unreachable ones. So, again as we inspect everyone of the objects here we take this, take this and take this return it to the storage pool and that ends the garbage collector operation. So, this gives you an overview of garbage collection there are obviously many varieties of garbage collectors which are in operation.

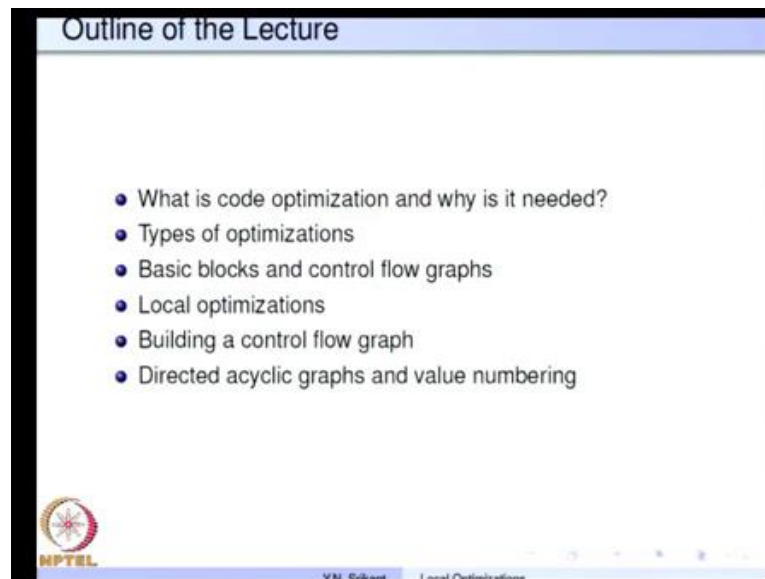
You know in the literature the concurrent garbage collectors, copy garbage collectors etcetera are all described. They are all useful in different circumstances, my point you know my aim was to give you an overview of garbage collection. So, with this the runtime system lecture ends so and we will continue with local optimizations.

(Refer Slide Time: 21:29)



So, welcome to part one of the lecture on control flow graph and local optimizations.

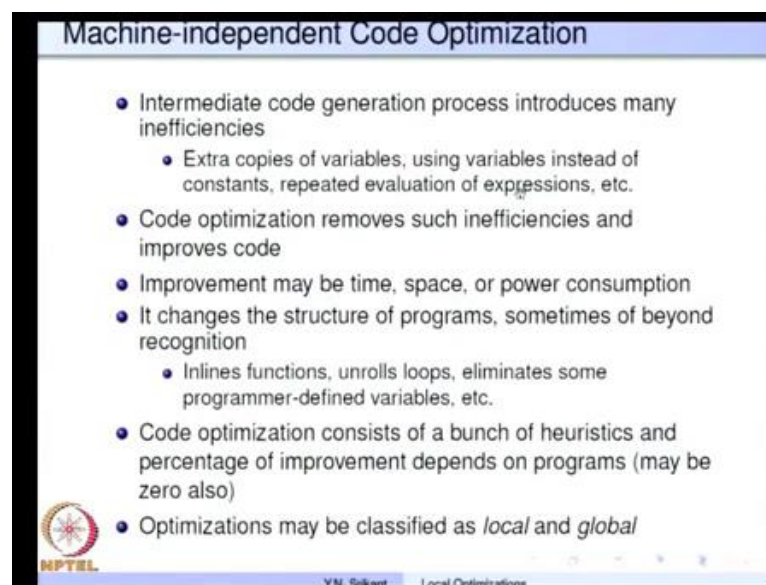
(Refer Slide Time: 21:39)



So, you know in the past lectures we have seen different phases of the compiler. For example, to begin with we saw lexical analysis, then we saw syntax analysis, then we discussed semantic analysis and after that we discussed you know intermediate code generation. And of course, we also saw the runtime support that is necessary. Now, before we generate machine code I mentioned in the early lectures, that there is something called machine independent optimization, which needs to be performed.

So, in this lecture we are going to see one type of machine independent code optimization called as the local optimization and the other types of optimization, will be discussed later. So, in this lecture we will see the reasons rather the definition of code optimization its purpose and then the types of optimizations. We are going to study what are basic blocks, what are control flow graphs the different types of local optimizations the procedure for building a control flow graph. And then of course, the methods of storing basic blocks efficiently using you know value numbering directed acyclic graphs etcetera.

(Refer Slide Time: 23:17)



So, what exactly is machine independent code optimization and why is it necessary. So, the intermediate we have discussed intermediate code generation in rare detail and we know that it is a fairly straight forward process. It introduces a large number of inefficiencies into the intermediate code that is generated. For example, whenever there is an expression a equal to b plus c , you know we would first do t_1 equal to b plus c and then do a equal to t_1 . Now, we have really introduced an extra copy of a in the temporary t_1 .

So, this is what I mean by extra copies of variables and using variables instead of constants. So, well there are many constants that we would like to use in the program, there is no straight forward way of using the may not be a straight forward way of using the constants in some cases. So, the intermediate code generator would generate an

assignment statement to assign a constant to a variable and then use that variable instead of the constant itself.

Then there are repeated evaluation of expressions so even though an expression such as a star b has not change its value. We may be evaluating a star b all over again within a short period of time, instead of trying to reuse the value which was computed earlier. So, these are all things which happen you know famous example, of this repeated evaluation is that i star 4 in the case of a of i plus b of i etcetera, etcetera. So, these are the sources of inefficiency in intermediate code.

The reason we did not bother about such inefficiencies is that we know code optimization would remove such inefficiencies and make the code better. Of course, improvement of the code may be these are only a minor examples, there are many more examples which we will see in the future. The improvement may be in time, in space or in power consumption. So, when we improve the time requirement which is probably what we do most of the time, we would be making the program run faster and faster.

When do we require improvement in space in other words we would like the program to take as little space as possible? So, in embedded systems memory is actually very expensive to incorporate so, we may have to pack all our programs into a very small memory. And in such cases, memory based optimizations to reduce the space requirements becomes very important. The same is true in embedded systems and a sensor network systems and so on. Power consumption becomes a very important factor in such cases.

So, embedded systems and sensor networks you know they run on small size batteries. So, it is necessary that the battery can be used to a you know, it can be used over a large period of time. To do this the power consumption of the program has to be minimized, if we simply make the program run faster you know. Of course, the power consumption will reduce because the program is now running for a shorter period of time, but there may be very subtle optimizations possible via you know by a compilers. Such as, reducing the voltage of the chip, switching of certain function units etcetera.

So, the compiler may be able to perform up you know optimizations keeping these features in mind. So, power consumption reduction is another type of optimization which is sometimes possible. So, the in general optimizations change the structure of programs

in fact sometimes beyond recognition. For example, optimizer's inline functions you know so the function call disappears the body of the function replaces the call.

It unrolls loops so instead of running the loop you know 300 thousand times, it can possibly unroll the loop 3 times and then say now, I will run it only 100 thousand number of times. So, why should it do such things the unrolling of loops etcetera will increase the opportunities available for instruction scheduling etcetera, and it may actually increase the parallelism in the program? So, the same is true for in lining of functions so these are all desirable optimizations, it may also eliminate some of the programmer defined variables.

So for example, in induction variable elimination the variable, which is defined as an induction variable may eventually get eliminated from the program. These effect of all this would be that we cannot use a debugger on optimized programs because the program has now changed and the mapping to the source code is no more possible. So, you know debugging an optimized program may not give out appropriate you know hints to the programmer at all. So, in a gcc for example, kind of stops all the optimizations if you choose the debugger option.

Code optimization in general consists of a bunch of heuristics so in other words, the percentage improvement is not guaranteed by any optimizer. The simplest example, would be you know if there is a set of statements which cannot be improved further then applying any number of heuristics on it will not improve the program. So, in such a case the code optimizer cannot do anything. So, you cannot guaranty the amount of improvement that a code optimizer provides, sometimes it may be 0 as well. Optimizations made be very broadly classified as local optimizations and global optimizations.

(Refer Slide Time:30:25)



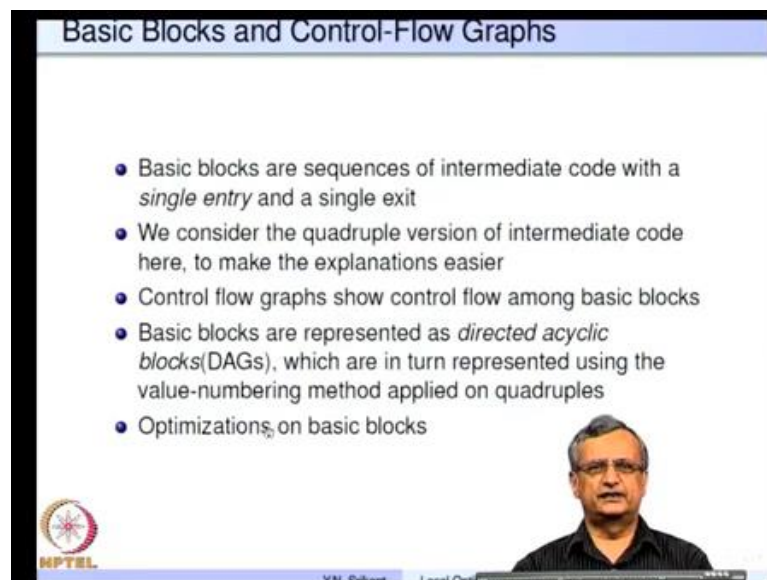
So, let me give you a few examples, of these optimizations all local optimizations are within what are known as basic blocks, we will know what these are in very in a very short period of time. To be very brief basic blocks are single entry, single exit areas of code and in such basic blocks whatever optimization we perform. The effect will not go beyond the basic block that is why they are called local optimizations. Whereas, the other type of optimizations are global optimizations so the effect of the optimization is on the whole procedure or even the whole program.

The some of the optimizations with the important ones are local common subexpression elimination. Then dead code elimination so what is dead code instructions that compute a value that is never used. That is really dead code elimination of such dead code and reordering computations using algebraic laws such as commutativity, associativity etcetera. So, these are all called as local optimizations we are going to see in this lecture how to perform such local optimizations.

There are the some of the very important global optimizations are listed here the global common subexpression elimination, which is similar to in effect as the local common subexpression elimination. Then there is constant propagation, constant folding, loop invariant code motion, partial redundancy elimination, loop unrolling, function inlining, vectorization, concurrentization.

There are a host of others as well more than another does not important optimizations which are possible and the gcc performs quite a few of them. So, we will work you know perform we will understand how to perform local optimizations in this lecture and then in the later lectures we will see how to perform global optimizations as well.

(Refer Slide Time: 32:33)



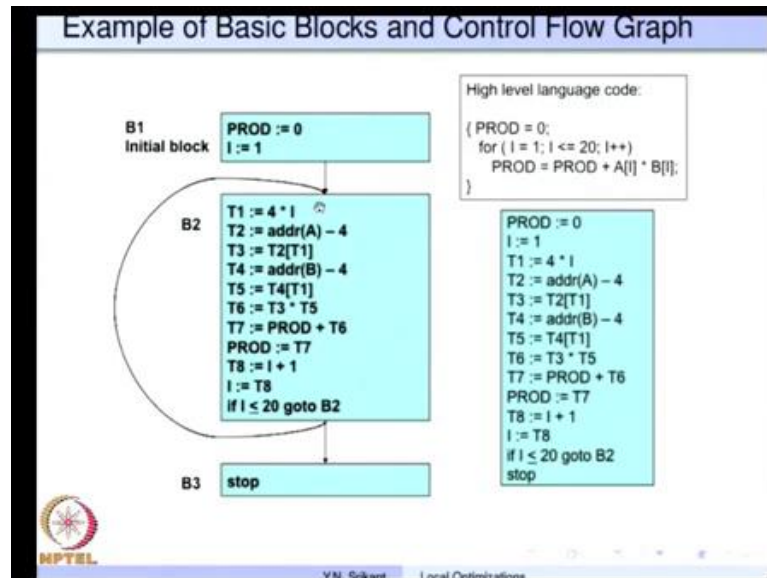
The slide is titled "Basic Blocks and Control-Flow Graphs" and contains the following bullet points:

- Basic blocks are sequences of intermediate code with a *single entry* and a single exit
- We consider the quadruple version of intermediate code here, to make the explanations easier
- Control flow graphs show control flow among basic blocks
- Basic blocks are represented as *directed acyclic blocks*(DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

The slide also features a video inset of a man speaking in the bottom right corner, an NPTEL logo in the bottom left, and a footer with the text "Y.N. Saha" and "Local Opti".

So, as I said local optimizations are all performed on basic blocks so we must understand the definition of basic blocks, how to construct basic blocks etcetera before we go to optimizations. So, basic blocks are sequences of intermediate code with a single entry and a single exit. So, let me show you an example before we consider further which we before we go further.

(Refer Slide Time: 33:01)

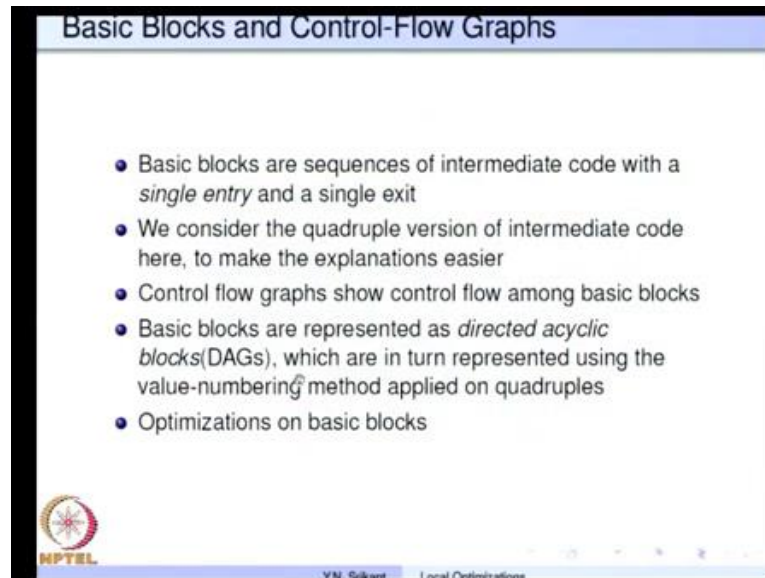


So, for example, this entire piece of code is one basic block so you can see that there is a single entry here and there is a single exit at this point as well. Similarly, these two statements together form a single basic block. So, here is the entry to the whole procedure and here is the exit from the basic block. So, but at the same time this piece of code which corresponds to the entire high level language program which is written here. So, this is a the simple dot product program is not a basic block, why so as we go on here is an entry then we go down.

And then, there is a goto which actually reenters the program you know so the b 2 part is the beginning of this t 1 so, here is the b 2 part. So, this is reentering the program again at this point so this is not a basic block, this whole thing has to be still carved into basic blocks. So, we consider the quadruple version of intermediate code here, we already saw that in the picture to make explanations easier.

And control flow graphs show control flow among basic blocks so this is actually a control flow graph. So, control flow between this and this block is shown by this arc, this and this block is shown by this arc and this is a loop again goes back to the beginning of the same block. So, these are all you know arcs in the control flow graphs and the entire structure is called as a control flow graph.

(Refer Slide Time: 34:55)



Basic Blocks and Control-Flow Graphs

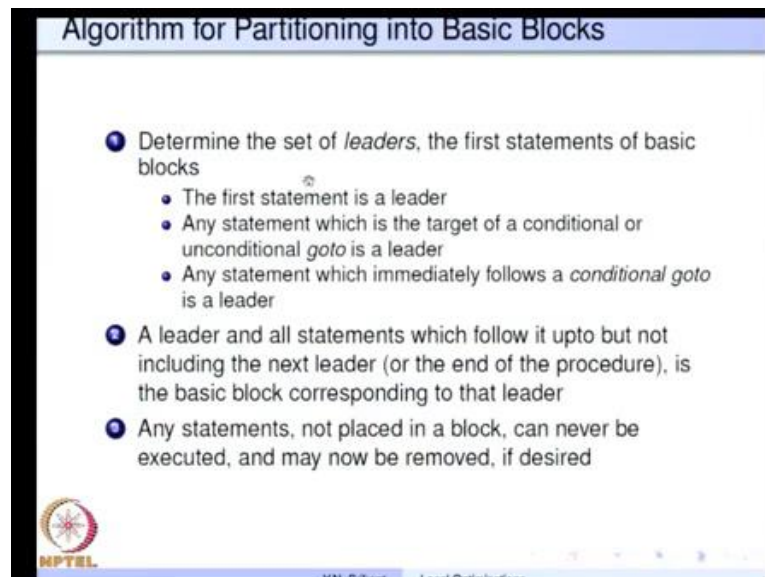
- Basic blocks are sequences of intermediate code with a *single entry* and a single exit
- We consider the quadruple version of intermediate code here, to make the explanations easier
- Control flow graphs show control flow among basic blocks
- Basic blocks are represented as *directed acyclic blocks*(DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

NPTEL

YN Sakari Local Optimizations

Basic blocks are represented as directed acyclic graphs so they do not have any loops within them. So, they are really directed acyclic graphs and we will see how to represent them using value numbering method applied on quadruples. And we will also see how to perform the local optimizations on basic blocks.

(Refer Slide Time: 35:20)



Algorithm for Partitioning into Basic Blocks

- 1 Determine the set of *leaders*, the first statements of basic blocks
 - The first statement is a leader
 - Any statement which is the target of a conditional or unconditional *goto* is a leader
 - Any statement which immediately follows a *conditional goto* is a leader
- 2 A leader and all statements which follow it upto but not including the next leader (or the end of the procedure), is the basic block corresponding to that leader
- 3 Any statements, not placed in a block, can never be executed, and may now be removed, if desired

NPTEL

YN Sakari Local Optimizations

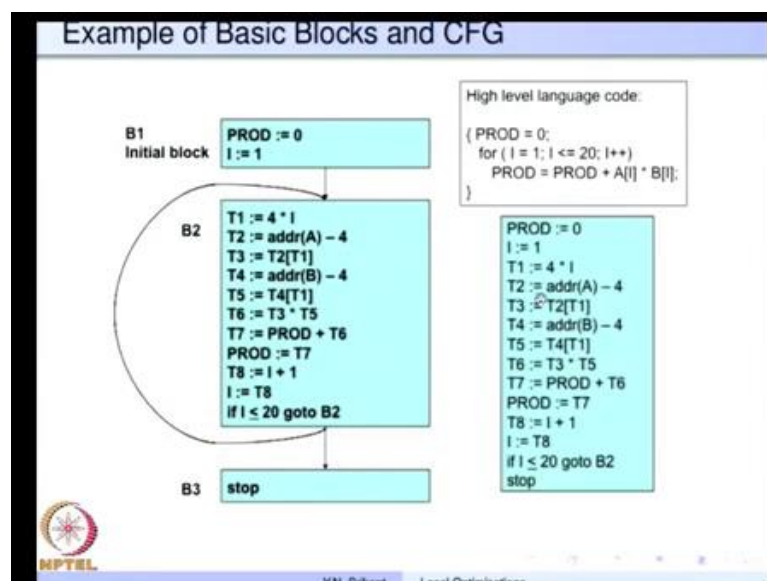
So, let us see how to carve out basic blocks from a piece of intermediate code. So, we must determine what are known as the set of leaders, that is the set of first statements of the basic blocks. And once actually get all the leaders in the program you know getting

basic blocks is very easy. A leader and all statements which follow it up to, but not including the next leader or the end of the procedure is the corresponding block, basic block corresponding to that leader so that is very easy.

So, the basic idea is to form leaders and any statements not placed in a block can never be executed and can be removed. So, this is the dead code elimination that we perform so if code does not get into a block then it can be it is unreachable and it can be removed. Now, how to get the leaders the first statement of the program is obviously a leader so you must begin somewhere so this is the first statement.

Then any statement which is the target of a conditional or unconditional goto is a leader. So, please understand this carefully we are not talking about the conditional goto statement for say, but we are talking about the target of the conditional statement. So, the target is an entry point to the basic block so that is the why it is a leader. And any statement which immediately follows a conditional goto is a leader.

(Refer Slide Time: 37:09)



So, let me show you how to carve out these basic blocks from this procedure. So, the first statement is a leader so this particular thing you know prod equal to 0 is a leader. Then we continue scanning we come here if i less than or equal to 20 b 2, b 2 of course, begins with t 1 so that is this right. So, this statement t 1 equal to 4 star i is a leader so we have set prod equal to 0 is a leader t 1 equal to 4 star i is another leader.

Then here is this is a conditional go to statement, the statement which follows it is also a leader so stop is also a leader. So, we have formed three leaders, why is it we say the statement following a conditional goto is a leader and not an unconditional go to. The reason is very simple a conditional goto has 2 exits one is going back of course, in this case. And the other one is if the value of i is greater than 20 the code falls through and executes stop so this is the other exit.

So, control can go either backwards or forwards that is why the next statement after the conditional goto is an executable statement. And that can be marked as a leader and of course, since the entry is the target is nothing but another entry into the program this will also be called as a leader. Now, suppose this were to be just go to b 2 instead of if i less than or equal to 20 goto b 2. Suppose, this statement was just goto b 2 then if you observe we will actually go into an infinite loop and we will never be able to reach this statement stop.

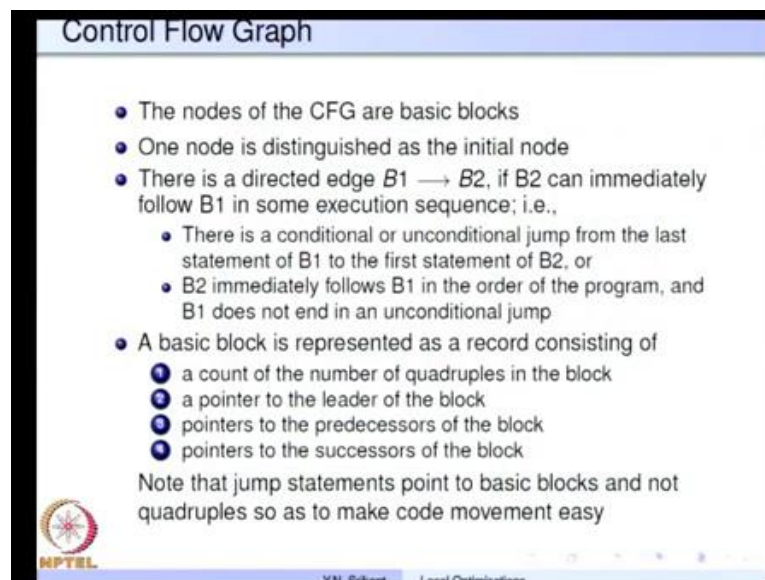
So, we should never mark this stop as a leader you know this is really dead code and it will get eliminated. Once, we have formed the set of leaders so this is a leader then this t 1 is a leader and stop is a leader. So, from prod to t 1 so not inclusive is the first basic block and from t 1 up to stop not inclusive is a second basic block. And from stop till the end of the program which is stop itself is the third basic block. So, this is how basic blocks are carved out of a program.

Now, to do a little of little bit of work on these basic blocks we must also add the arcs into between these basic blocks in order to form the control flow graph. So, how do we add these arcs of course, the first thing is we look at the you know look at the basic block, look at the last statement in the basic block. Then if it is not a you know unconditional goto statement then the statement following it can be actually taken as the target of this particular arc.

So, in this case we have i equal to 1 which is not a goto and therefore, we mark a you know we place an arc between these two to indicate the flow of control. Similarly, we goto the last statement of this particular block it is not an unconditional goto, if it were then this would have been dead code. So, this is conditional goto so the next statement will be executed if the condition is false.

So, we place an arc from here to here indicating that control flow can happen here as well. Then of course, we have this arc to show that this is the target of this you know conditional goto so, we place the arc between this point the end of the basic block to the target b 2 so this is how we place arcs to to form a control flow graph.

(Refer Slide Time: 41:24)



Control Flow Graph

- The nodes of the CFG are basic blocks
- One node is distinguished as the initial node
- There is a directed edge $B1 \rightarrow B2$, if $B2$ can immediately follow $B1$ in some execution sequence; i.e.,
 - There is a conditional or unconditional jump from the last statement of $B1$ to the first statement of $B2$, or
 - $B2$ immediately follows $B1$ in the order of the program, and $B1$ does not end in an unconditional jump
- A basic block is represented as a record consisting of
 - ① a count of the number of quadruples in the block
 - ② a pointer to the leader of the block
 - ③ pointers to the predecessors of the block
 - ④ pointers to the successors of the block

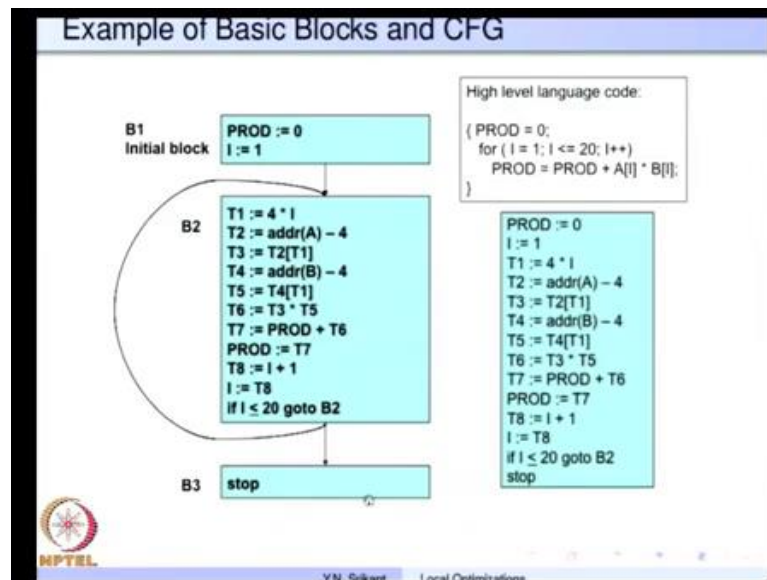
Note that jump statements point to basic blocks and not quadruples so as to make code movement easy

NPTEL Y.N. Sarda Local Optimizations

So, this is what I just now explain the nodes of the control flow graph are basic blocks, one node is distinguished as the initial node so this is the procedure entry. There is a directed edge b 1 to b 2 if b 2 can immediately follow b 1 in some execution sequence. So, this is what I just explain that is there is a conditional or unconditional jump from the last statement of b 1 to the first statement of b 2. So, here from the last statement of this to the first statement of this.

And this of course, is not at all a goto so from here to here b 2 immediately follows b 1 in the order of the program and b 1 does not end in an unconditional jump so, this is the other case. Now, how we do represent a basic block, we represent it as a record the record has a count of the number of quadruples in the block a pointer to the leader of the block. Then we need the predecessors and successors of the block as well so with this we can actually form the control flow graph as well. So, the jumps to points to the basic blocks and the not quadruples and this makes code movement easy so let me explain why this is so.

(Refer Slide Time: 42:55)

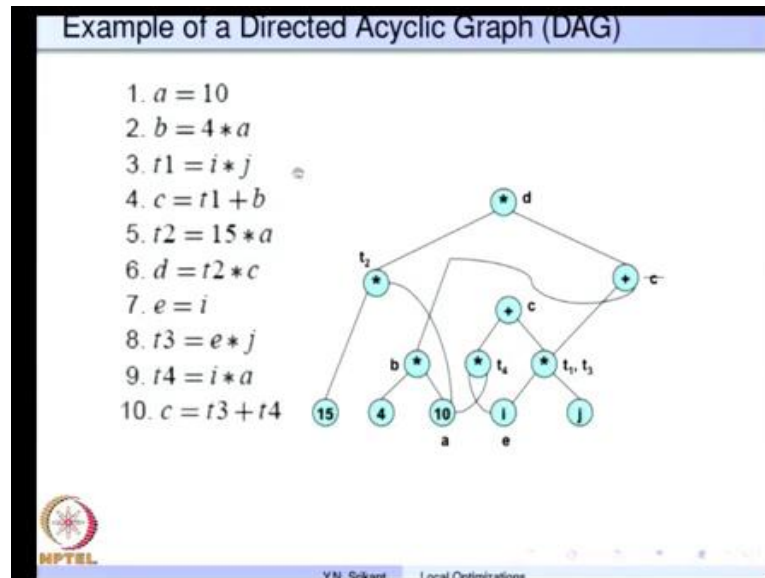


Just imagine that this statement if says if i less than or equal to 20 go to 3 because this is statement number three really one, two and three right. Suppose, we really you know add a few more statements here right some statements from here may move to this statement, this particular basic block during some optimization. In such a case, what happens is the the number of this particular quadruple the first quadruple in the basic block b 2 the it is number will change, it may become 5 or 6 let us say.

So, if we had maintained goto 3 here we would have had to change that as well you know this number will have to be changed to 5 or 6 appropriately. Whereas, if we simply say goto b 2, where b 2 is nothing but a pointer to the basic block record. Then we know that in spite of the changes made to the number of quadruples in this basic block this pointer does not change. So, there is really no harm no change that is necessary.

So, this makes code movement easy so we do not have to worry about changing any numbers if we move code from here to here or to some other place. Now, let us understand how to represent the basic block using directed acyclic graphs. So, what I said here you know this is the extra statement corresponding to a basic block we still have all the quadruples in the basic block and we must actually represent them using directed acyclic graphs. This is Meta information corresponding to a basic block.

(Refer Slide Time: 44:51)



Here, is a basic block there are ten statements in this basic block. So, observe that there are no jumps here it is just one block of statements. So let us understand what this structure is this is obviously a directed acyclic graph there is no loop here. And let us understand how to build this dag from this set of quadruples so the procedure is what I am going to describe now.

So, here is a quadruple $a = 10$ so we form a node a , a node containing the value 10 and we attach a label to it called a . And then there is an assignment statement $b = 4 * a$ now we know we search the dag how to do the search we will see later. We search the dag find that there is a node with label a in it so, this node b can you know this star node can be created with left child as 4 and the right child as this a . And we attach a label b to it so we have formed this computation tree as far as these two statements are concern.

Third one says $t1 = i * j$ so similarly, we have i here we have j here we create the new nodes i and j . Then we create a new node star attach these two arcs and attach a label $t1$ to it. So, the notation that I have used is the first time that we create a node the value is put inside, whether it is a variable or a constant and any extra variables are attached as labels. Then we have $c = t1 + b$ so we must form actually a node called plus and label it as c .

So, this node plus will have t_1 as its left child and it will have b as its right child. So, we can search the dag find out the appropriate nodes corresponding to b and t_1 and establish these links. Then we have t_2 equal to $15 \text{ star } a$ so we have t_2 here, so 15 is a new node that we want to create, a , is already present. So, we can create a star node and attach the links and label it as t_2 , then we have d equal to $t_2 \text{ star } c$ which is very similar.

So, we search for t_2 we find it, then we search for c we find it and then we form the star node and label it as d . We have e equal to i we find i so, we do not have to really do any extra you know creation of node we just attach a label e to it. Then we find t_3 equal to $e \text{ star } j$ so, e is here already available. Then you know t_3 is this so e and i are the same j is also here already. So, we simply do not create another node for t_3 we attach it to the star node. So, this has two labels now t_1 and t_3 .

Then the next statement is t_4 equal to $i \text{ star } a$, so we have t_4 here this is a new node that star is a new node that needs to be created, i is already there a is also already there. So, we create a new star node and attach a label to it as t_4 . The last one is important it says c equal to $t_3 \text{ plus } t_4$ so, we find you know t_3 already available. And we also find t_4 which is available here, but the point is the variable c is being reassigned. So, there was already a c variable here, so we need to actually kill the occurrence of the old variable it is not relevant anymore create a new node plus attach a label c to it.

So, what are the advantages of this particular approach well, if you observe carefully you know we never created the node for t_3 all over again? So, t_1 corresponds to $i \text{ star } j$ and t_3 even though it is written as $e \text{ star } j$ because of the assignment e equal to i it is really $i \text{ star } j$. That means, the expression $i \text{ star } j$ is now both t_1 is available in t_1 and t_3 you know we would have recomputed the $i \text{ star } j$ two times if we had used this code. Whereas, if we use the dag representation we are not going to recompute an $i \text{ star } j$ all over again it is available in t_1 . So, we will simply use t_1 in place of t_3 in our entire program. So, this is an example of common subexpression elimination.

Then the second advantage for example, here we have b equal to $4 \text{ star } a$ so we of course, have created a node for star for the example I showed that. We established links to 4 and 10 here a is 10 of course, and instead of doing that when we search for a , since we get a value 10 and 4 is actually a constant as it is. We could have performed this multiplication 4 into 10 as 40 , replaced this entire structure by having one you know node b with a

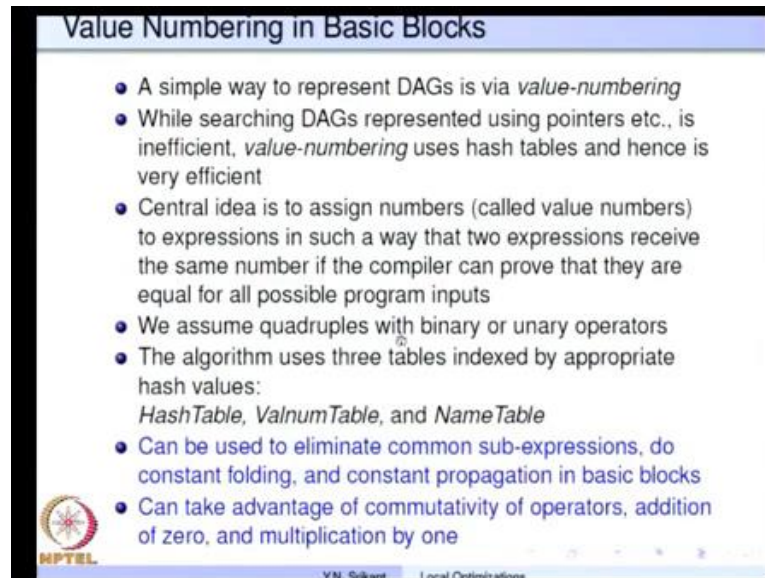
value of 40. This entire computation could have been eliminated. So, this is known as constant folding. We have propagated the value of a from here to here, so that is constant propagation. Then we can evaluate 4×10 as 40 so that would be constant folding.

So, there is possibility of constant propagation, constant folding and also common subexpression elimination if we use the directed acyclic graph representation here. The only difficulty is if we really build a directed acyclic graph, using the pointers and links as we have seen as we see in this picture this is the way we build trees you see. So, if we build such a structure physically by establishing these links and nodes and so on and so forth. It is not of much use really let me explain why, so at some point we want to search for you know say b or something like that so, we want to search for a are even better.

There is no simple mechanism to search for a, we will have to start at the root of the directed acyclic graph. Search the possibly the entire dag systematically and then at some point in this search we may we will end up getting this a. So, in the worst case you will search the entire dag this is a you know very expensive process. If we had some thousand statements in the basic block, then you know searching the entire dag of 1000 nodes is very time consuming.

So, construction you know of the dag and trying to find common subexpressions or nodes in the dag by exhaustive search of the directed acyclic graph is a time waste, you know is a time consuming procedure and is a waste of time. If we use links and tree like structures like this, this is the only way whereas, if we avoid using these links represent this entire directed acyclic graph using hash table structures. Then you know the entire operation becomes much, much faster.

(Refer Slide Time: 54:05)



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
HashTable, *ValnumTable*, and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

NPTEL

So, this process is known as value numbering and it is used in the construction of basic blocks. So, a simple way to represent DAG's is via value numbering and value numbering uses hash tables. So, searching for expressions and variables is very quick, very time efficient. The basic idea is to assign numbers to expressions and these numbers are called value numbers. So, when we assign such numbers when we want to check whether two expressions are the same, we simply compare their value numbers. If the value numbers are the same then the two expressions are equivalent, they will produce the value at all times. So, this is the idea behind value numbering. Again, we are going to assume quadruples with binary or unary operators the algorithm will use different types of tables. For example, for expressions it uses hash tables and then for variables it uses valnum tables and for constants it uses name tables.

So, these you know value numbering methods can be used to eliminate common subexpressions, do constant folding then they can be used to perform constant propagation in basic blocks. And they can of course, be used to you know take advantage of the commutativity of operators, addition of 0, multiplication of by an 1 and so on and so forth. So, at this point we will stop the lecture and continue with details in the next lecture.

Thank you.