Compiler Design

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

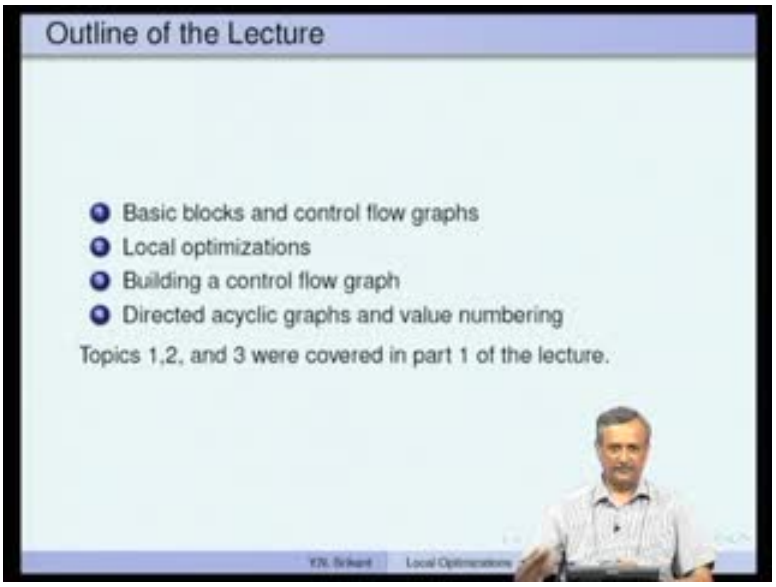
Module No. # 03

Lecture No. # 07

Local Optimizations-Part 2 and Code Generation

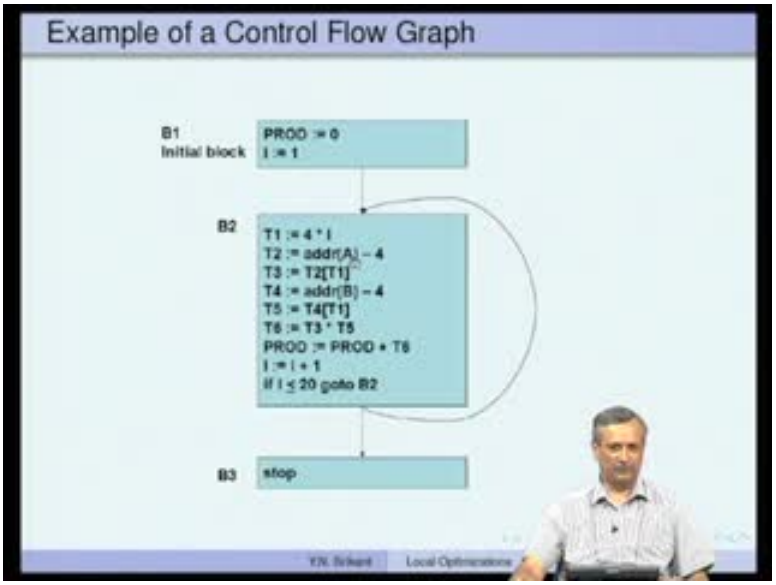
Welcome to part 2 of the lecture on Local Optimizations.

(Refer Slide Time: 00:20)



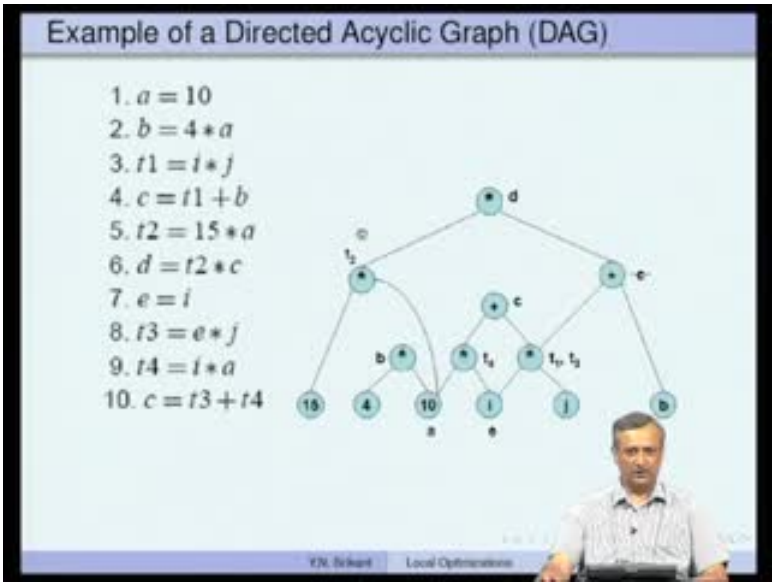
In the last lecture, we learnt about basic blocks and control flow graphs. Few of you know the optimizations as examples; then how to build a flow graph. Today, we will continue in the same direction to see what a directed acyclic graph is and understand value numbering.

(Refer Slide Time: 00:46)



Just to recapitulate from the last lecture, here is an example of a control flow graph. So, these are the basic blocks. Each basic block is a single entry and single exit structure. There are arcs between the basic blocks to show the control flow within the program. The contents of each of these basic blocks are quadruples and that is a form of intermediate code. These basic blocks are built by identifying leaders. They are actually optimization which can be carried out on this basic block and we are going to learn from now on.

(Refer Slide Time: 01:34)



Here is an example of a Directed Acyclic Graph – DAG. Here is a set of 10 quadruples, a small basic block. A basic block can be represented in the form of a Directed Acyclic Graph and here is the DAG representation of this particular basic block. So, why are we studying this representation of a basic block? First of all, this is the conceptual structure that we need to build in order to perform local optimizations. It very clearly tells us what the capabilities of such an optimizer are.

Let us go a quadruple by quadruple and see how this particular DAG structure is built. The first one is - a equal to 10. So, here is a node a, you know a node containing 10 is labeled as a. So, this is a way and we are going to do it, whenever there is a value or whenever there is an operator star plus etc. It is going to be placed inside the node as text. There is a label attached to each of these nodes, which will tell us what is the name associated with that particular value or operator. So, a equal to 10 corresponds to this particular node with the value 10 inside and the label a is outside. The next one is b equal to 4 star a. So, here is a node containing a value 4. The node star connects 4 and a and it is named as b.

The next one is t1 equal to i star j, which is very similar. There is a node i and there is a node j. These names are also put inside because there is no value attached to them now. Then there is a star, which shows that i and j are connected. The value and the label associated with the node star is t1 to begin with, t3 arrives later. c equal to t1 plus b is similar, t1 is used as one of the children and plus b is used as another child. It is labeled as c and the source striking of c has an extra meaning, which will be cleared later on.

Now, t2 equal to 15 star a; a is reused, 15 is created, star is created and a label t2 is attached to this particular star. Now, d equal to t2 star c. Again, we take t2, c and then you know these two are related by star and that is actually labeled as d. There is another statement e equal to i. So, there is a node already with a value i or a label i and there is no modification of i, which has been carried out. So, the value of e will be the same as the value of i. Therefore, we attach the label e to this particular node i and nothing else, no other action is taken.

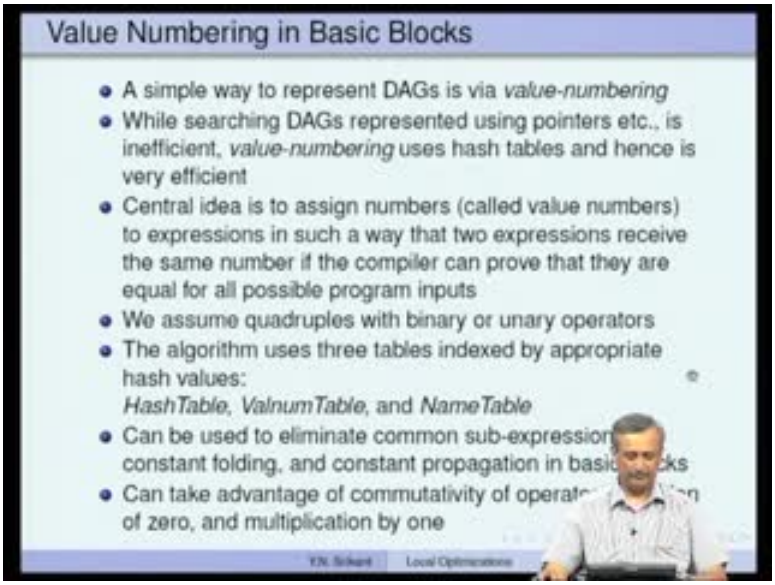
Now, t3 equal to e star j. So, e star j is actually discovered as i star j because e is nothing but a label and the node i. Therefore, i star j and e star j are identical. I star j; small sub tree already has a label t1. Therefore, t3 is also attached to same list of labels. So, t1 and

t3 both correspond to this particular star (Refer Slide Time: 05:32). There is no further node creation, which is carried out. So, this is an example of how common sub-expressions are actually discovered. So, i star j was an expression which was computed. Now, e star j is nothing but i star j and it is also discovered as a common sub-expression. So, we do not have to compute it again. We just used the previous expression and this is indicated by attaching to label t3 to this particular node star.

Now, t4 equal to i star a. So, this is really at this part the picture. So, there is a and you know this is i. Now, there is something special, which happens. Here, it is stated as i star a and what is taken here is a star i. So, this is possible because we assume at this point that the star operator is commutative. So, by using commutative d, we checked whether i star a was not present.

We discover that a star i is present and therefore, you know a and i are present. We created a node star and attached a label t4 to it and c equal to t3 plus t4 is very similar. Where is t3 and t4? This is an example, where no value have been assigned to c and whole value in quadruple number 4 is no longer going to be used. So, this is overriding that particular value and this old value is scaled. The node label c is deleted and a new label c is attached to this particular pass. So, this DAG hopefully shows examples, where there is a reuse of nodes and common sub-expressions are discovered and so on. Now, let us see how to implement this type of a scheme to carry out some of the local optimization.

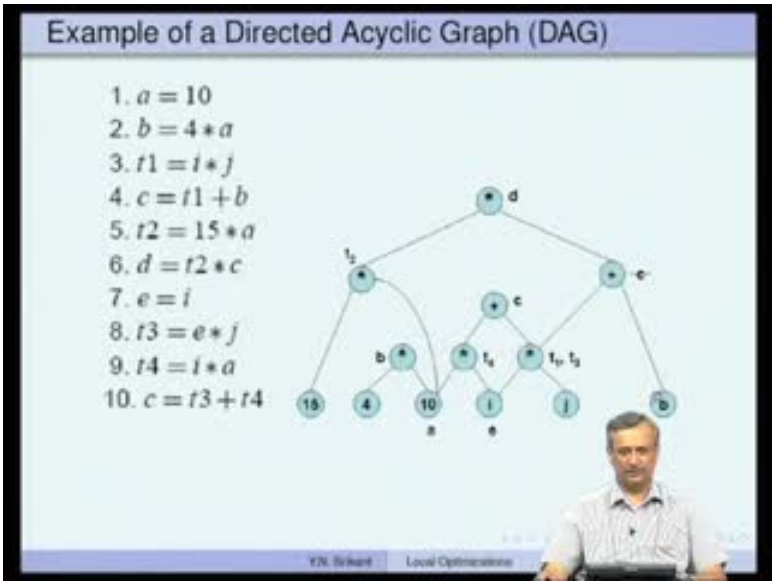
(Refer Slide Time: 07:46)



Value numbering is a simple way to represent DAGs. While searching the directed acyclic graphs, if we represent those using pointers etc, it is extremely inefficient. For example, if you are searching for a node in the DAG, it is not possible to search for the node without exhaustively searching the entire DAG. So, in practice and implementation, you know optimizations on DAGs. We use the value numbering in scheme rather than in a linked representation of DAGs. The value numbering scheme uses hash tables and therefore, it is very efficient in searching the graph itself.

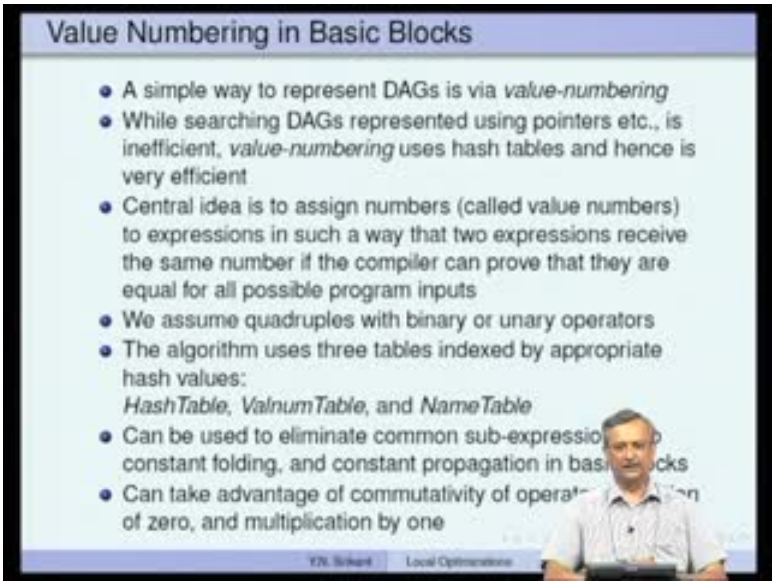
What is the central idea? The central idea of value numbering is to assign numbers called value numbers. So, these are assigned to expressions and of course, they are also assigned to single variables, in such a way that two expressions receive the same number, if the compiler can prove that they are equal for all possible program inputs. So, we saw this happen in the previous picture and only thing is we had not assigned any value numbers.

(Refer Slide Time: 09:11)



For example, here, you know i star j was already present. So, e star j was discovered as a common sub-expression. We will soon see that this particular expression, i star j and another expression e star j will both receive the same value numbers.

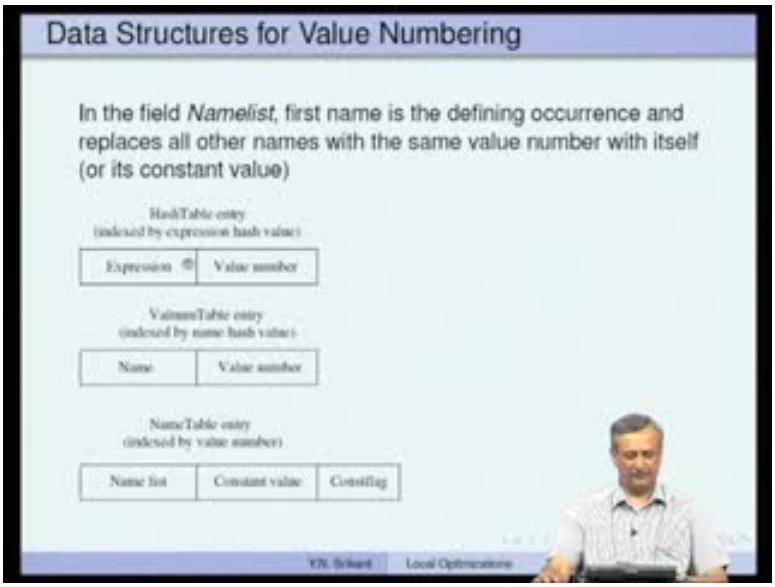
(Refer Slide Time: 09:29)



So, what we assume is quadruple with binary or unary operators. We also permit copy operations, assignment of constants and so on to variables. There are 3 tables, which are used by this particular algorithm and they are all hash tables. So, I will show some

pictures of these data structures very soon. The first one is called a hash table, the other one is called as a value number table and the third one is called as a name table.

(Refer Slide Time: 10:14)



Let us see some pictures. So, here is a hash table entry, it is indexed by expression hash values. In other words, we take an expression hash and the expression will give us some particular value index. Go to that particular index in this hash table. If the expression which is present there is the same as the one we are searching, then there is success. If there is no success, we insert that particular expression into that index. So, in each one of these entries, we store the actual expression itself and the value number. Value number is nothing but an accounter. So, we initialize it to 1 and then we go on incrementing it. We need a new value number and as the expression hash table as a hash table, which is storing expressions

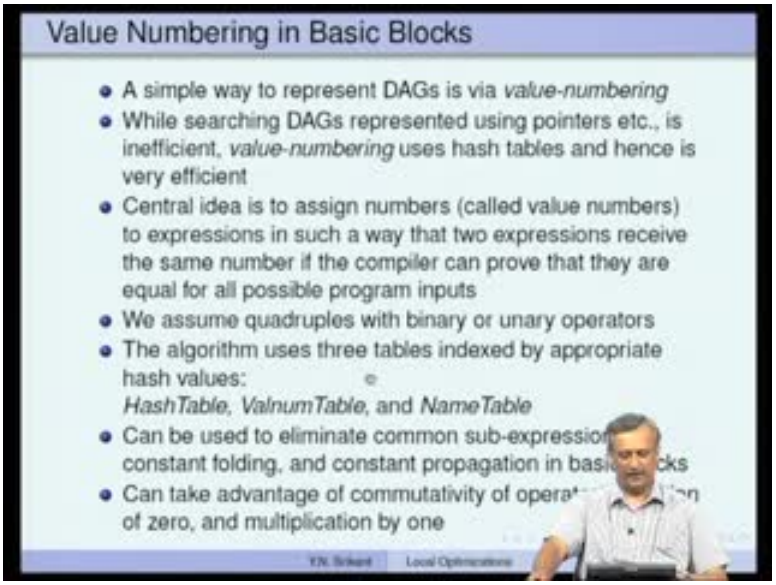
The second one is called a value number table. A value number table is shown here. Each entry has a name and a value number this is indexed by the hash value of the name. Why do we have a separate a value number table? Why do not we store names also in the hash table itself? The hashing function for expressions will have the 2 variables of the expression assuming that it is a binary operator. So, it will have parameters; the 2 operators or operations, sorry the operands. Let us say in a star b, we take a and b and then the operator, star has to be incorporated into the hashing function. So, depending on the 3 values – a, star and b; the index value produced is going to be different. The same

expression hashing function cannot be used for names, where you do not have any operator and you do not have a second name. So, we use a different table for storing names and this is a value number table. For each name, you hash it and then you go to that particular index and pick up the value number, if the name is already present there.

The third one is called as a name table. So, name tables are shown here and the entries are shown here. An entry is shown here, there is a name place, there is a constant value and there is a constant flag. A name actually produces its index by the value number. So, the names and value numbers are given. A value numbers is used to index into this particular table called the name table. So, inside the name table, we store the constant value of that particular name. If it is a constant, then set the constant flag to 1. If the name that is stored here is not a constant, then this (Refer Slide Time: 13:32) is set to 0. This particular field has no relevance.

What is this list here? So, in the field name list, first name in the list is always the defining occurrence of that particular name. It replaces all other names with the same value number with itself or its constant value. So, if there are many names, which are equivalent and we saw this happen in this particular figure. Here, t1 and t3 are equivalent. So, here e and i are equivalent and so on. So, in such a case, we are going to have a list of such names here, which are equivalent. The first one is the defining occurrence, which will be used in place of all other names in that particular list. If it is a constant, we are going to use that value of that constant in place of all these names.

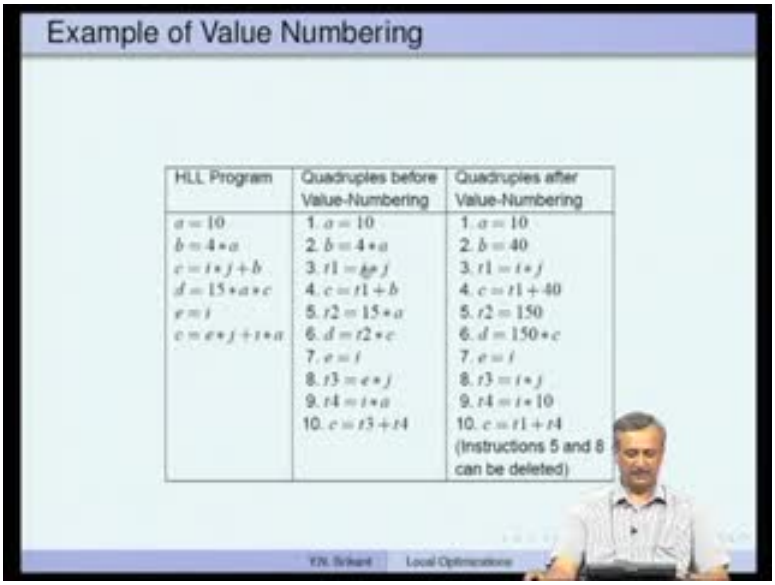
(Refer Slide Time: 14:28)



The value numbering scheme can be used to eliminate common sub-expressions, constant folding can also be performed and constant propagation can be carried out in basic blocks. So, elimination of common sub-expression is something which I already mentioned. Constant folding is arriving of expressions involving only constants. Constant propagation is - if there is a assignment, which is a equal to 4. Wherever a is used, 4 is going to be propagated to that point and that is called constant propagation. We will see how to do these things with value numbering.

Value numbering can also take advantage of the commutativity of operators, which I briefly mentioned a few minutes ago with respect to that directed cyclic graph representation. It is audition of 0 because a plus 0 is a itself. Multiplication by 1 a star 1 is a. These are the 3 simple algebraic laws that are taken care by value numbering in basic blocks.

(Refer Slide Time: 15:35)



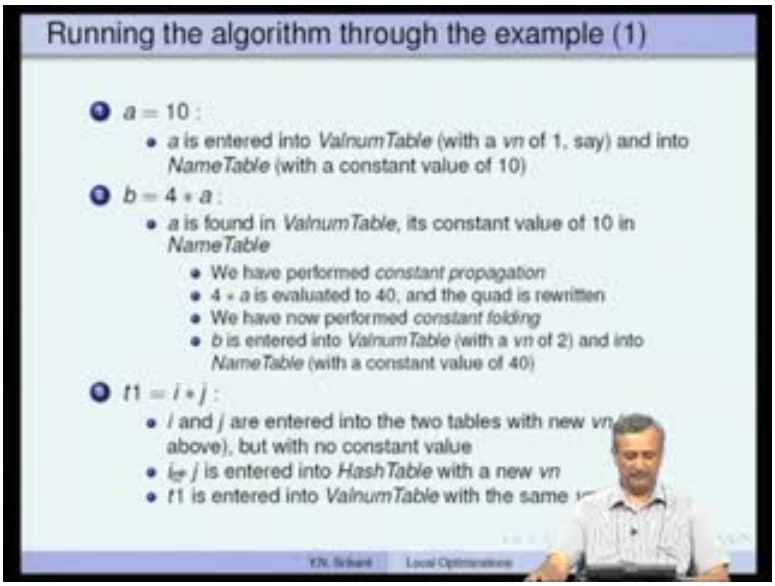
Let us take an example in a step by step fashion and see how it works out. Here, is a High Level Language program: a equal to 10, b equal to 4 star a, c equal to i star j plus b, d equal to 15 star a star c, e equal to i and c equal to e star j plus i star a. So, if you look at this here, a equal to 10. So, actually, if we observe, it is the same program that we saw here and there is no difference.

So, only thing we have considered is the quadruple version directly. Here is the high level language version. So, i star a plus b is broken into 2 quadruples, 15 star a star c is also broken into many quadruples. Similarly, this is also broken into many quadruples. So, what is the set of quadruples after the optimization? Let us understand that and then see how the optimization is carried out. So, a equal to 10 remains in 4 star a, b equals to 4 star a, a is 10. So, we want to replace it by b equal to 40, t1 equal to star j remains as it is. C equal to t1 plus b, b is replaced by 40, which is now being calculated as a constant t and t equal to 15 star a becomes t2 equal to 150 because a is a constant value of 10.

D equal to t2 star c becomes d equals 150 star c because t2 is 150 e equal to i remains as e equal to i. t3 equal to e star j becomes t3 equals i star j. We find that t1 equal to i star j and t3 equal to i star j are the same one. So, we can delete this particular instruction number 8 and always use t1 in its place. Further, t4 equal to i star a becomes t4 equals to i star 10. C equals t3 plus t4 becomes c equals to t1 plus t4 because t1 and t3 have the same value. As I said, 8 can be deleted and t2 equals to 150 and the value of 150 has

already been propagated to all the other places. For example, t2 star c is the only place, where t2 is used. So, this quadruple is not needed any more, 5 and 8 can be deleted. We have deducted that i star j is a common sub-expression. We have done constant folding by evaluating these constant expressions. We have done constant propagation by replacing these bs and as by constant. So, let us see how all these can be done.

(Refer Slide Time: 18:36)



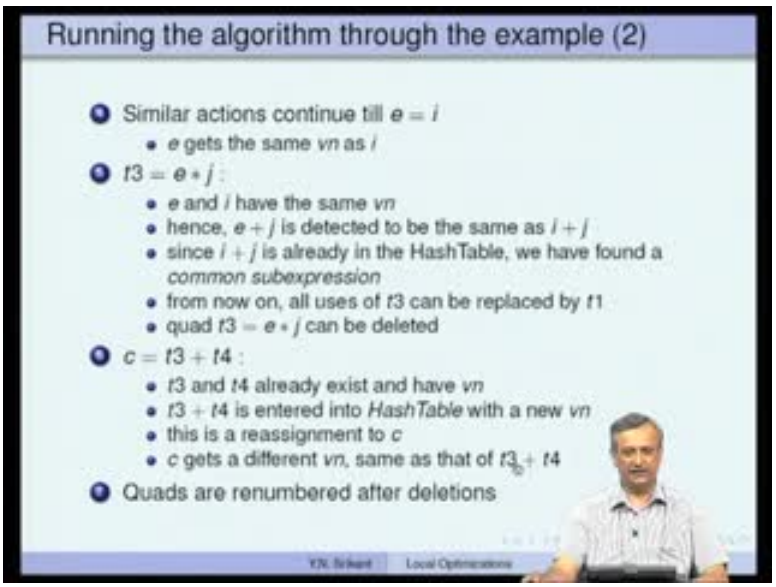
The first quadruple is a equal to 10. Now, a is entered into the value number table. Let us say, a value number of 1 is the beginning. So, we have set vn, the counter value as 1. It is also entered into the name table with a constant value of 10. So, here, we hashed enter into value number table and the value number is set as 1. Using that value number index into the name table, set the constant value as 10 because they have set the constant flag as 1

Second quadruple: b equal to 4 star a. Now, when we search for a in the value number table, which is a variable. We find that it is already present and its value is 10 and that value can be found on the name table by using this value number in the value number table. So, we have performed constant propagation because we find that a is 10 and we can replace it. Here, 4 star a is evaluated to 40, this is already said. The quad is rewritten and this is the other action.

You have now performed constant folding and b is entered into the value number table with a value number of 2. This is the different value number in the name table, using the

same value number of 2 with a constant value of 40. So far, we have entered these two into the appropriate tables, t1 equal to i star j is similar. I and j are not present in the table. So, they are entered into the tables. No value numbers are created and there is no constant value attached to either i or j because they are not constants.

(Refer Slide Time: 20:49)



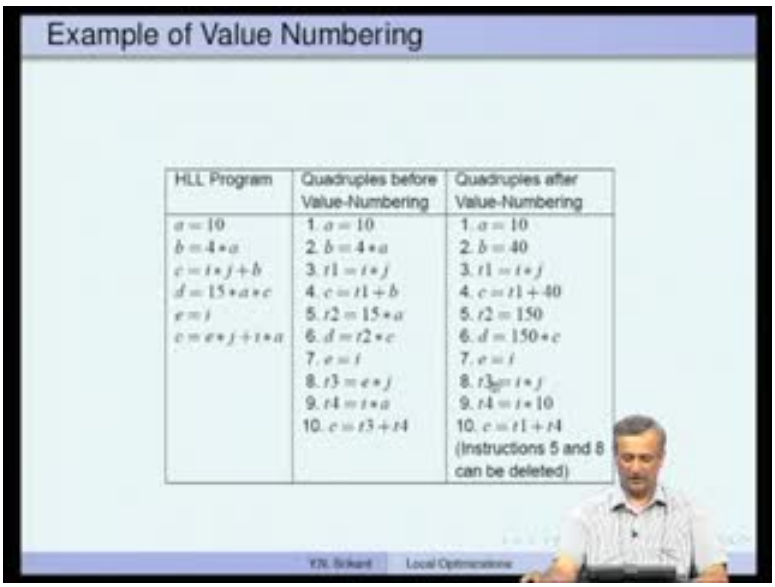
Now, i star j is entered into the hash table with a new value number. So, we hash this and go to that index and store it. Assign a new value number because i star j was not present before. T1 is entered into the value number table with the same vn, as i star j. So, we assigned some value number to i star j. The same value number is assigned to t1 because these 2 always carry the same value until this point. In turn, e equal to I and i is already present. Thanks to i star j, i was introduced into the table some time ago. So, we actually get it from the table.

Find the value number and then assign the same value number to e also. Whenever we get e, we replace it with i. So, what we do? This is actually entered into that name table along with i. I is the first defining entry and then e is going to be attached to the same list. In processing t3 equal to e star j, we find that e and i have the same value number search. You get the same value number e plus j and it is nothing but i plus j, sorry e star j and this plus is a mistake. So, e star j is same as i star j and since i star j is already in the hash table, we have found a common sub-expression. So, please read this plus as star.

From now on, all users of t3 can be replaced by t1. so, t1 was already present in the hash table and there is no need to create another sub-expression called i star j. Quad t3 equal to e star j is nothing but i star j and it can be deleted. So, we found that i star j is present and t1 is actually attached to it with the same value number. Now, we have the same value number attached to t3. Therefore, t1 and t3 are 2 occurrences of the same expression with the same value number. So, whenever we get t3, we can replace it by t1. So, t1 and t3 will be entered into the name table. T1 has a defining occurrence and t3 following it.

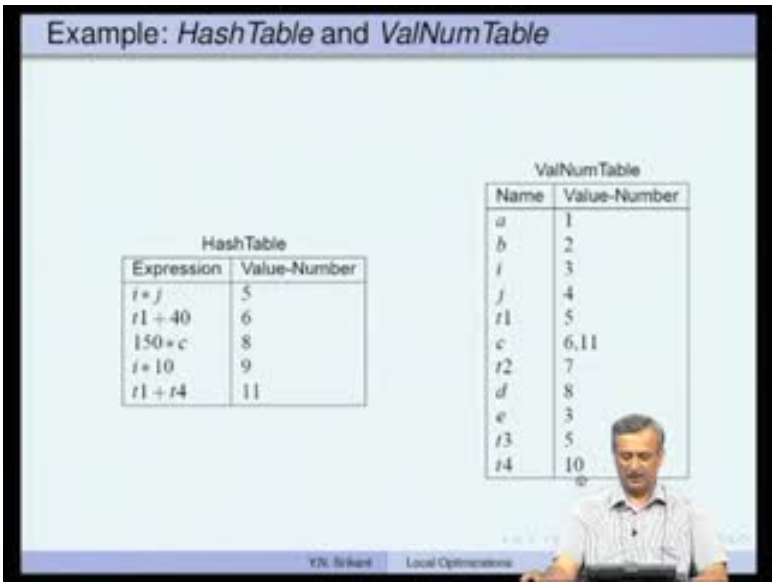
Now, c equal to t3 plus t4 and t3, t4 are already in the table. They have a value number 2. So, t3 plus t4 is entered into the hash table with a new value number. So, we hash it and enter. This is not a common sub-expression, which is already present. This is a reassignment to c and this is what we need to be careful about. So far, this is not the third quadruple and you will know about the quadruple a little later. So, this is a sixth quadruple, t3 plus t4 was not there. So, we have entered it, but this is a new c and this is not the same old c. C was defined earlier, but now, c is redefined. So, c gets a different value number and it is the same as the t3 plus t4. So, the old is disabled, its value number is not used anymore and it is rewritten with the new value number.

(Refer Slide Time: 24:06)



These quadruples are renumbered after the deletions. We get an optimized tough quadruple as we saw here. So, you know 5 and 8 go away. 5 is gone, 8 is gone and then we make 6 as 5 and 7 as 6 etc.

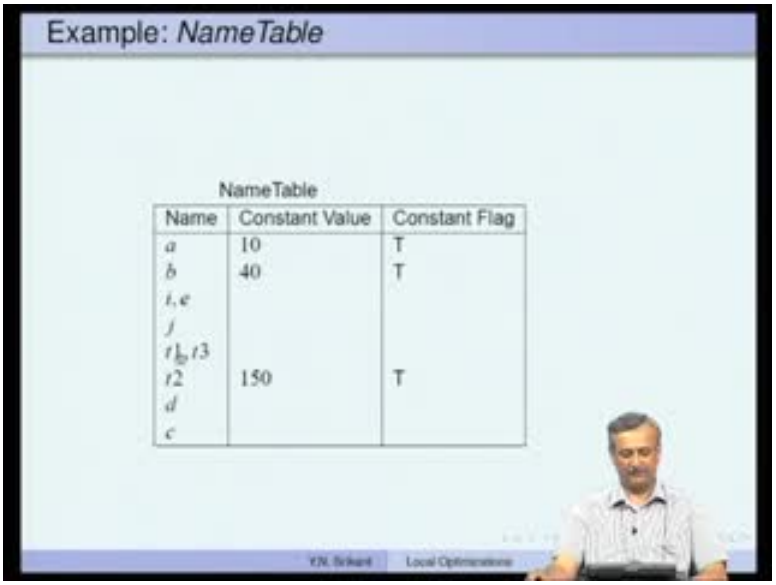
(Refer Slide Time: 24:19)



Here is the hash table and value number table at finishing time. So, see that i star j has value 5, t1 plus 40 is 6, 150 star c is 8, i star 10 is nine, t1 plus t4 is 11. So, these are the various expressions that we have found with distinct value numbers.

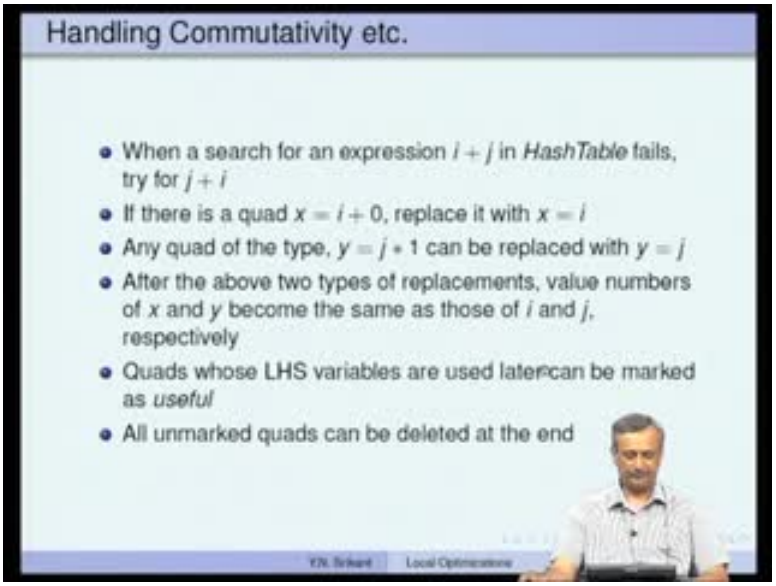
Value number table is interesting and a, b, i, j, t1 and c are all given 1, 2, 3, 4, 5, 6. Then t2 as 7, d as 8, e did not get any value number and has got the same value number as 3. Then t3 did not get any value number, it got the same value number as t1. Then t4 got new value number 10 and the second time, c got a value number 6, 11 and 6 was overwritten.

(Refer Slide Time: 25:32)



Here is the name table at the end. So, a is stored with the constant value 10 and const flag 2, b is stored with 40 and 2, i and e are not constant and so there is nothing. Here, j is not a constant, t1 and t3 are not constants. So, please see that i and j, i and e are together. Here, i is the defining occurrence and replacement for e will be i. See whether, if replacement of t3 will be t1 itself, t2 is 150 and 2, d and c are not constants. So, this is how we do value numbering, create these tables, find common sub-expressions and so on.

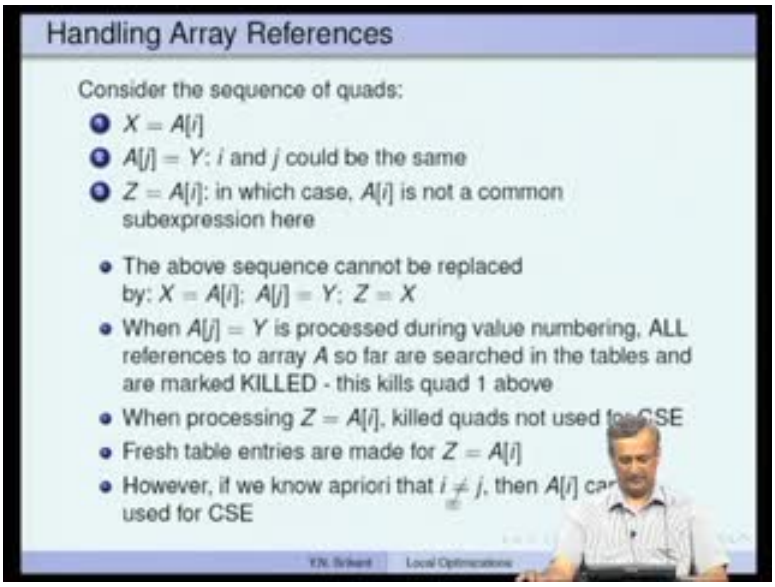
(Refer Slide Time: 25:46)



Now, let us see how to handle commutativity. So, I already mentioned this commutativity in the directed acyclic graph representation. So, when we search for a expression i plus j in the hash table. Let us say, this search fails, for example, plus is commutative on integers and not for floating point numbers. So, let us assume that commutativity holds.

Now, i plus j is not available, we search for j plus i. So, j plus i is present and we already saw this happened before, if there is a quadruple. So, this is using commutativity and exploiting commutativity, if there is a quadruple x equal to i plus 0. Then it is not necessary to keep as it is. Do the arithmetic with plus and it can be replaced with x equal to y. Perhaps, from now on, we can use i in place of x and finally, even x can be deleted.

(Refer Slide Time: 27:40)



Here, we are using an algebraic law, a equal to a plus 0. Any quadruple of the form y equal to j star 1 can be replaced with y equal to j and this is again a simple law. Now, a equal to b star can be replaced by a equal to b. After the above 2 types of replacements, value numbers of x and y become the same as those of i and j. So that is a very simple thing to see quadruples, whose LHS variables are used later. It can be marked as useful and all unmarked quadruples can be deleted at the end. So, this is the implementation detail and there are more complications, which come in these arrays.

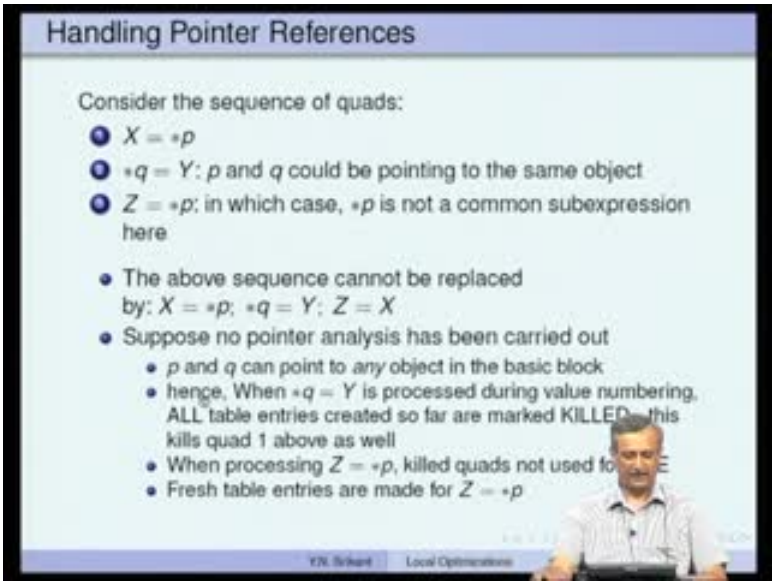
Let us consider a simple sequence of 3 assignments, x equal to A[i], A[j] equal to Y and Z equal to A[i]. So, if you look at this particular statement, A[j] equal to Y. I and j are

runtime variables and they can take any value. So, we have no ideas what the values are. So, it is possible that i and j could be the same.

If they are same by some chance, then you know A[i].is not a common sub-expression anymore and syntactically, this A[i] look the same. So, I could have possibly z equal to A x. If we have only looked at the syntax, this particular statement A[j] equal to Y. If i and j are the same, it should be interpreted as A[i] equal to Y, in which case, there is a reassignment of value to A[i]. The previous value, which was assigned to X cannot be used as a value of A[i] anymore. We need to retain these 3 statement as how they are. So, this is a problem. How do we handle it? Whenever there is an array expression, we actually do not try reusing the value of that array expression at all. So, the above sequence cannot be replaced by x equal to A[i], A[j] equal to Y and Z equal to X because of the reasons as I mentioned just now. So, this is processed as it is.

Here is an indexing assignment operator. So, we hash this particular expression - A[i], enter it into the table. X is taken as a defining occurrence which gets the same value number as A[i]. When you process A[j] equal to Y, all references to array A, so far are searched in the tables. We need to search exhaustively and there is no other mechanism. They are marked as killed and this kills the quadruple number 1 here. So, there could have been a verse such as A[i], A[j], A[k], A[x] etc. All are deleted or rather marked as killed and so that their value numbers are not reused from this point onwards. Y, this particular j could have been any one of those values. We do not know which value it is and it could be there or not, but we cannot take a chance when processing zeal to A[i]. Killed quadruples are not used for common sub-expression elimination. So, fresh table entries are created and it was Z equal to A[i]. There is a new value number, which is assigned to A[i]. However, if we know apriori that i not equal to j, so that these two are never identical. Then A[i] can be used as a common sub-expression in this particular case, but this is very rare. So, we may have to kill them.

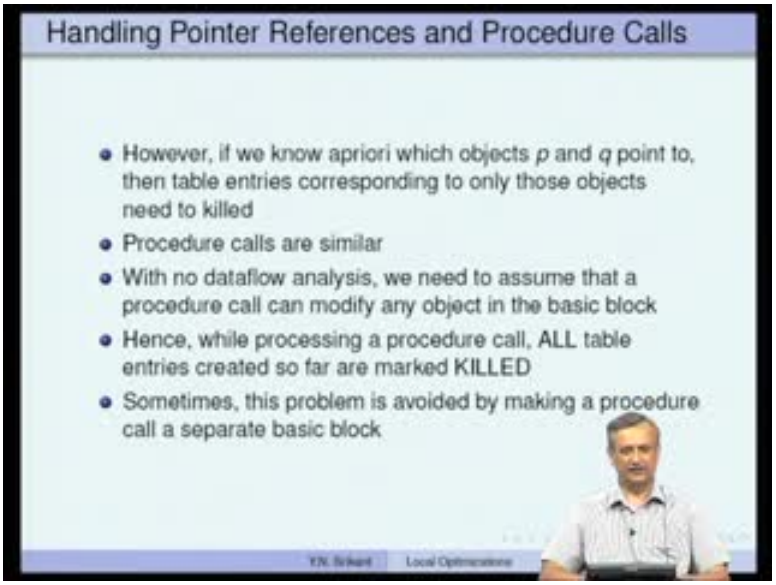
(Refer Slide Time: 31:12)



Pointers are very similar. So, consider X equal to star p, star q equal to Y and Z equal to star p. So, p and q could be pointing to the same object. You do not know at runtime that this can happen. If it has happened, then Z equal to star p is not a common sub-expression anymore because a new assignment is being made to star q. It is not X equal to… you know X was pointing to what p was pointing to. X contains star p and X contains object where p was pointing to. Now, star q contains new object Y and q points to a new object called Y. you know q has a value possibly p itself and so this should be read as star p equal to Y.

Now, p points to some other objects and therefore, Z equal to star p cannot become Z equal to X. This would be wrong and it is very similar to array case. So, if pointer analysis has been carried out, p and q never point to the same object. If that fact is known, then we can proceed with the same CSE here, but otherwise, p and q can point to any object in the basic block. Hence, when star q equal to Y is processed during value numbering, all table entries created so far are marked as killed. So, this kills the quadruple number 1 here. When we are processing Z equal to star p, killed quadruples are not used for CSE and fresh tables entries are made for Z equal to star p.

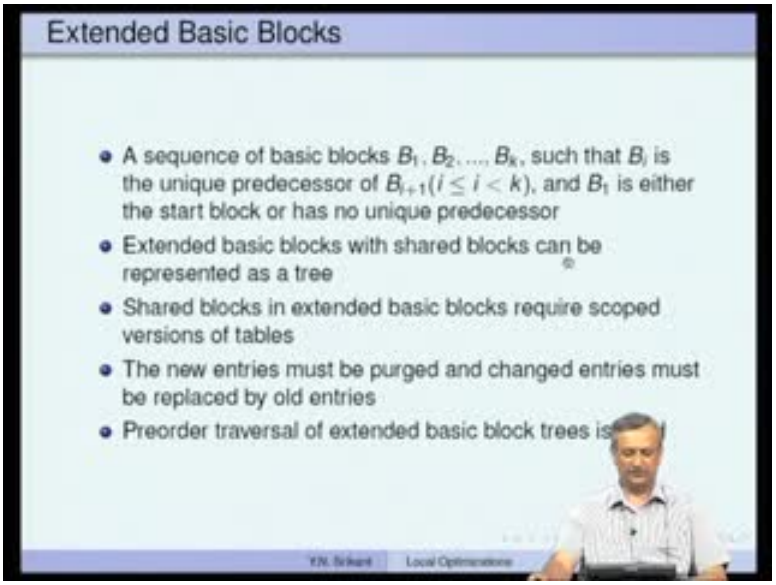
(Refer Slide Time: 33:07)



Something similar to that happens when we have procedure calls. So, if we do know apriori which objects p and q to, then table entries corresponding to only those objects need to be killed. So, this I mentioned already. So, what happens in procedures with no dataflow analysis of the program? We need to assume that a procedure call can modify any object in the basic block. So, if the procedure is called, it has side effects. So, it can assign to global variables and it can assign to parameters which have been passed by reference.

We have to assume that any variable in the basic block is modified because we have not carried out any analysis. We have no information about any variable. Hence, while processing a procedure call, all table entries created so far is not visible anymore and nothing is useful anymore. Sometimes, this problem is avoided by making a procedure call as a separate basic block. So, we eliminate this problem and we avoid it by making a separate basic block for a procedure call. So, it does not kill any other variable and since the procedure call is in its own basic block. This is a fairly simple thing to do. So, analysis becomes much simpler.

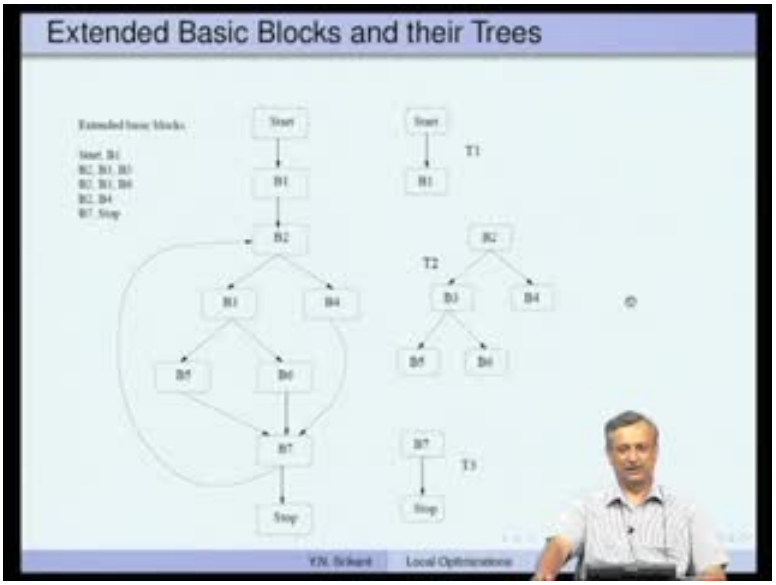
(Refer Slide Time: 34:46)



Now, we looked at basic blocks. Sometimes basic blocks are small and so what happens is all the techniques that we have applied may not be so effective. There is a way to extend whatever we did so far with a value numbering to a sequence of basic blocks. Let us see what these are.

A sequence of basic blocks B1 to Bk, such that Bi is the unique predecessor of Bi plus 1 for 1 less than equal to i; less than k. B1 is either the start block or has no unique predecessor. So, such a sequence of basic blocks is called as an extended basic block. So, extended basic blocks with shared blocks can be represented as a tree. So, let us look at some examples.

(Refer Slide Time: 35:46)



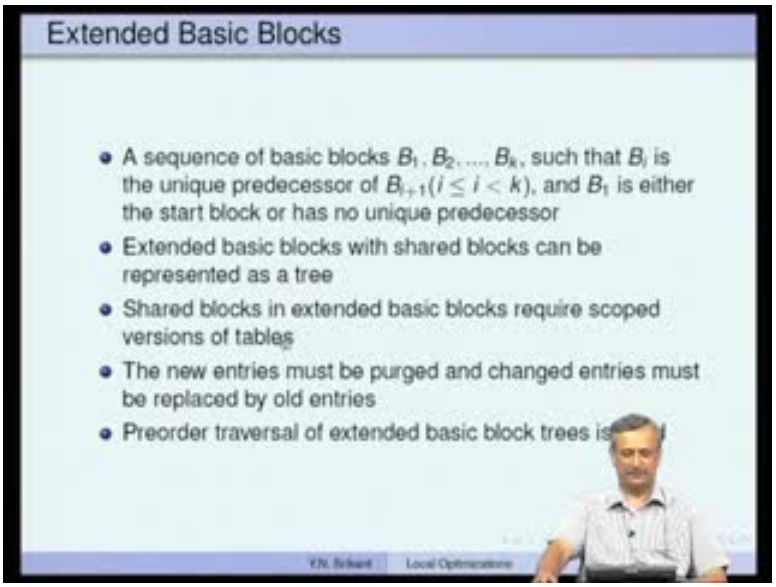
Here is a control flow graph. Star B1, B2, B3, B5, B7. This side is B6, B7 and this side is B4, B7 and then stop. So, the extended basic blocks are start and B1. When you go to B2, it has 2 predecessors and it does not have a unique predecessor. So, we stop our extended basic block at B1 itself. Then you consider B2, B3 and B5. So, if you try adding something else, for example, B7, it has 2 predecessor. So, we really cannot have that in the same basic block.

So, B2, B3 and B6 is another extended basic block. This is a sequence and we could not have added B2, B3, B5 and B6 because B5 and B6 are not in the same sequence. So, B2, B3, B6 are all… B5 is not the predecessor of B6 and vice versa. So, we cannot add this in the same extended basic block. So, B2, B3, B6 forms another extended basic block. Then B2, B4 forms one more extended basic block. Finally, B7 is stopped to form a basic block of the extended basic block.

Now, these 3 – B2, B3 and B5; B2, B3, B6 and B2, B4 actually share basic blocks. So, B2, B3 is shared with B5 and B6. B2 is shared between B3 and B4. So, there is a lot of sharing that happens in these and because of this, B2 is shared in all of them. Actually, it is B2, B3 and B5; B2, B3, B6 and then B2, B4. Hence, B2 is shared in all of them. Now, such a tree representation can be used gainfully to do our value numbering. Extend the benefits of value numbering to such basic block trees. So, what we do is very simple. We

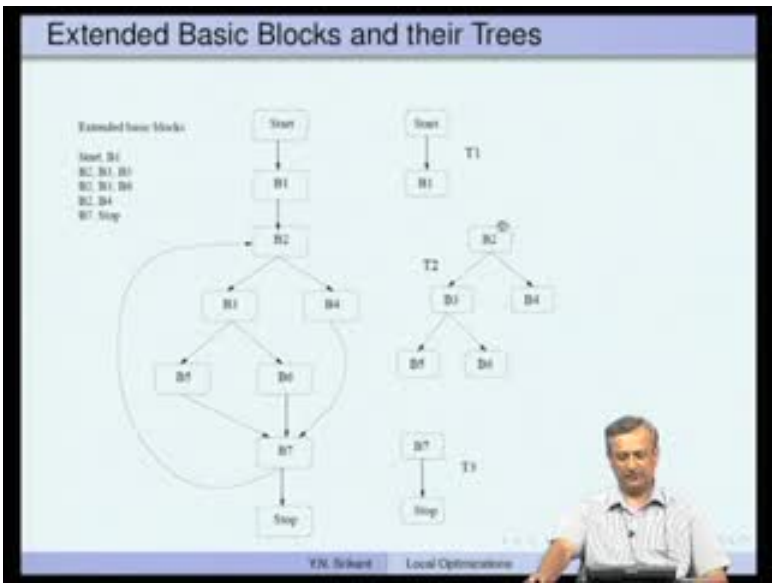
really need to use the technique, which was used in the single table construction for programming languages with nested scopes.

(Refer Slide Time: 38:17)



Shared blocks in extended basic blocks require scoped version of our tables. I will explain what these are. The new entries must be purged and changed entries must be replaced by old entries as we go on. Preorder traversal of extended basic block trees is used here. So, let us see what happens.

(Refer Slide Time: 38:43)



Let us consider this particular set of extended basic blocks. So, we start our value numbering with the basic block 2. It is usually done with the expressions. Variables are entered into various tables and they are filled. Then we take up B3 without destroying any of the tables built in B2. It is easy to see that we can use all the common sub-expressions and constants of B2 in B3 also. There is no problem at all and so we can keep the entries of B2 and reuse them in B3.

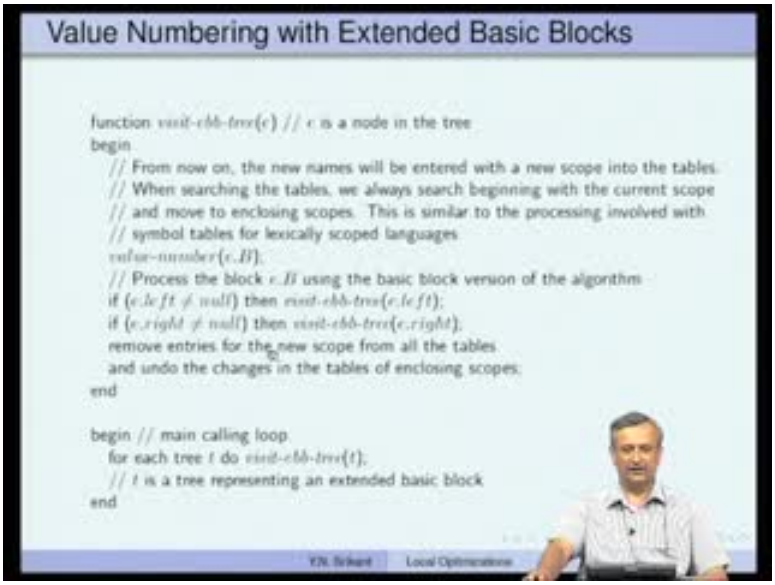
Now, when we go to B5, something similar happens. We can reuse the entries of B2 and B3 with no difficulty at all, once B5 is completely processed. Possibly, we have discovered new common expressions and constant propagation folding etc. It has been carried out using B2, B3 and also B5 is completed. Now, we have to get back to B3 and start processing B6. This is the preorder traversal, but the entries that we actually created during B5 cannot be made available during the processing of B6. We need to have the entries of B2, B3 and the entries of B5 must be eliminated. This is the reason why we require scoped versions of these tables.

We start a symbol table and we use something similar to scoped symbol tables. We have a set of tables at scope equal to 1. When we come to a new level, we actually establish tables at scope equal to 2. Here, we establish tables at scope equal to 3. Whenever we want to search these scoped tables, we actually search for the current tables first, then in next enclosing scope and so on, up the tree.

When we finish processing B5, all the tables scope will be equal to 3 are removed and a new table with scope equal to 3 is started for B6. Now, we use entries of B2 and B3 in B6. Finally, when B6 is completed, its entries are all deleted. We go to B3, when we need to return to B2. So, the entries of B3 scope equal to B2 are also removed. A new set of entries with scope equal to 2 corresponding to B4 are created and this is processed. Finally, we returned to B2 after deleting the entries corresponding to B4. So, this is how scoped hash tables are used in order to do value numbering on these extended basic blocks.

As you can see, we reuse entries from B2 and B3 in B5 and B6. There is a lot more scope for discovering common sub-expressions, finding constants, doing constant at propagation and so on.

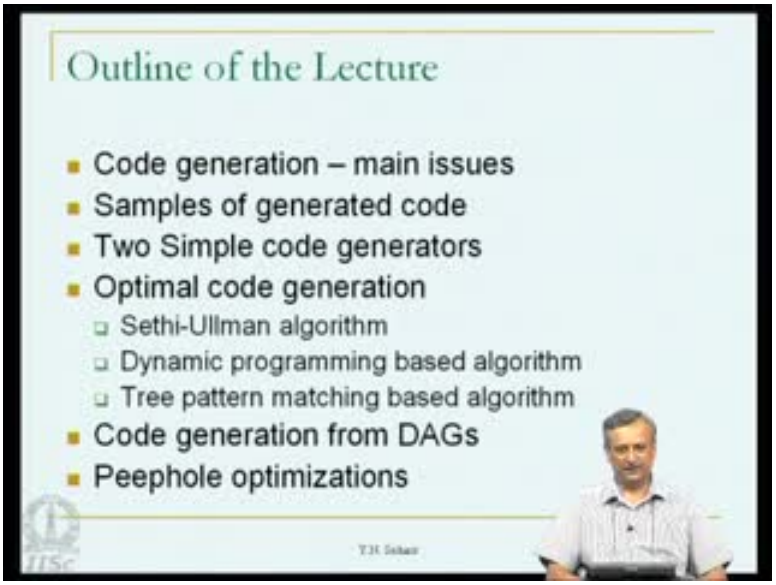
(Refer Slide Time: 42:04)



Here is a simple algorithm for doing value numbering. So, when we visit extended basic block tree and e is a node in the tree. So, it is to begin with the root of the tree. From now on, the new names will be entered into enter with a new scope to the tables. When searching the tables, we always search the beginning with the current scope. This is something I already mention. Moving to enclosing scopes, this is similar to the processing involved in the symbol tables with lexically scoped languages.

So, value number e, B is the call to the function value number to process the basic block B. Now, e dot B is the basic block and e dot left not equal to null, then visit left sub tree e dot right not equal to null. Then it is the right sub tree and this corresponds to the preorder traversal. Once the left and right sub trees are completed, remove the entries for the new scope from all the tables and undo the changes in the tables of the enclosing scopes. So, this is the function called visit-ebb and which is actually called for each tree representation of the extended basic block. So, this brings us to the end of the lecture on Local Optimizations.

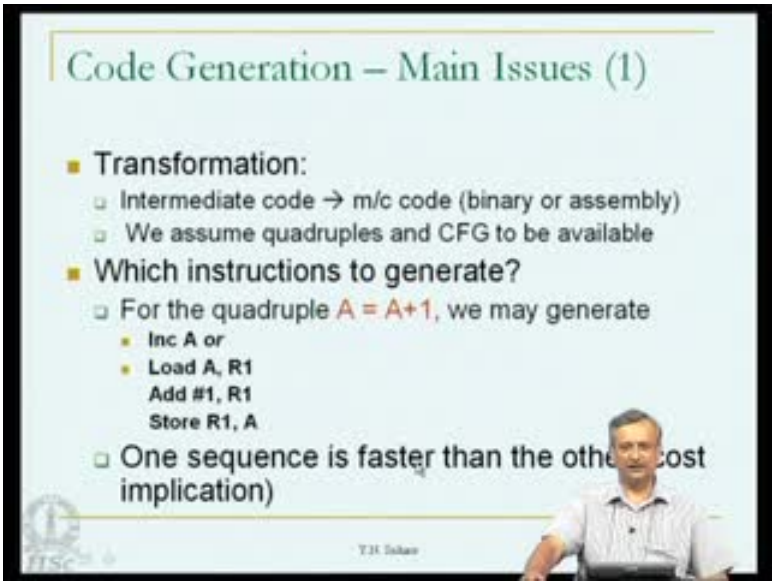
(Refer Slide Time: 43:43)



Welcome to the lecture on Code Generation. So, in this set of lectures, we are going to look at the main issues in machine code generation. Some samples of generated code two schemes for generating simple codes. So, these are simple code generators. Machine code generator is a very fascinating area with several optimal code generation algorithms. We are going to look at a couple of algorithms of this kind. One of them is the historically famous Sethi-Ullman algorithm and then there is a slightly more complicated dynamic programming based algorithm.

We are also going to look at a practical tool called iburg, which uses tree pattern matching and dynamic programming in its implementation. We are going to look at the method of generating code from directed acyclic graphs. This problem is mp complete. So, we need some heuristics to solve this problem. Finally, we will talk about peephole optimizations, which are machine dependent optimizations.

(Refer Slide Time: 45:03)

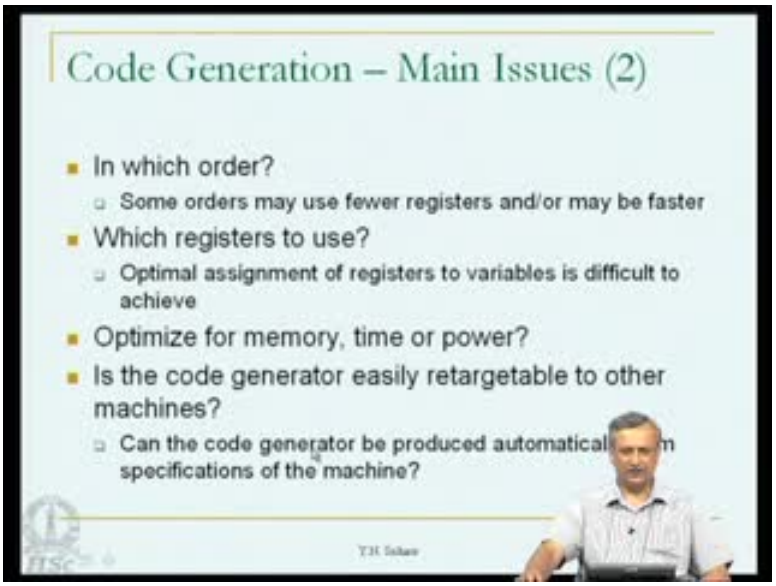


What are the main issues in machine code generation? Machine code generation is a transformation. So, it transforms intermediate code to machine code. Machine code is either binary representation or assembly language representation. If it is a binary representation, all the machine code, which is generated for the program can be sent to the linker and then the loader. Whereas, if the machine code is in the form of assembly, it has to be sent to the assembler, which in turn generates binary and then the binaries are sent to linker and the loader. The difficulty of generating code is the same, whether we generate machine code in binary or machine code in assembly. It is just that if we want to generate binary, the job of doing assembly is also in some way incorporated into the machine code generator itself.

Let us assume that we generate assembly because it is easier to explain the scheme. In such a case, let us assume quadruples as our intermediate form and let us also assume that the control flow graph is available to us. The first issue is about instructions that we generate. So, for example, let us say, the quadruple is a equal to a plus 1. A very simple instruction, such as increment A can be generated or we can say load A into the register R1. Add the constant 1 to R1 and store the register R1 in A. So, either 1 instruction or 3 instructions will obviously increment and it is much cheaper as everybody can realize. If there is no increment instruction in the instruction refer tool of the machine, we need to generate this particular sequence. The point here is every machine is different and its instruction set is different. Therefore, we cannot assume the existence of a particular type

of instruction in every machine. Our algorithm for the code generation will have to be cheaper to every type of machine. What we know is that every machine will have instructions to perform any task and that is all we really know. So, here, one of the sequences is faster than the other and such a case, we say that one of the sequences are cheaper than the other.

(Refer Slide Time: 48:16)



Second issue is - in which order should we generate code? It is possible that some of the orders use fewer registers and some of the orders may also be faster than the others. So, the code generator to some extent must evaluate different orders and then choose the best. Obviously, it cannot enumerate all orders and check them. It uses other mechanisms to check a few of these orders. Finally, choose the one, which is the cheapest.

One more issue is - which registers should we use in code generation? Some of the registers are reserved for particular types of use. For example, the program counter, the stack pointer are specific registers, which should not be used for any other purpose. This is the norm in programming, but there are many registers in machines. These registers can be used to store any variables. So, which registers to use and which variables should we put in those registers etc is the problem of register allocation in general.

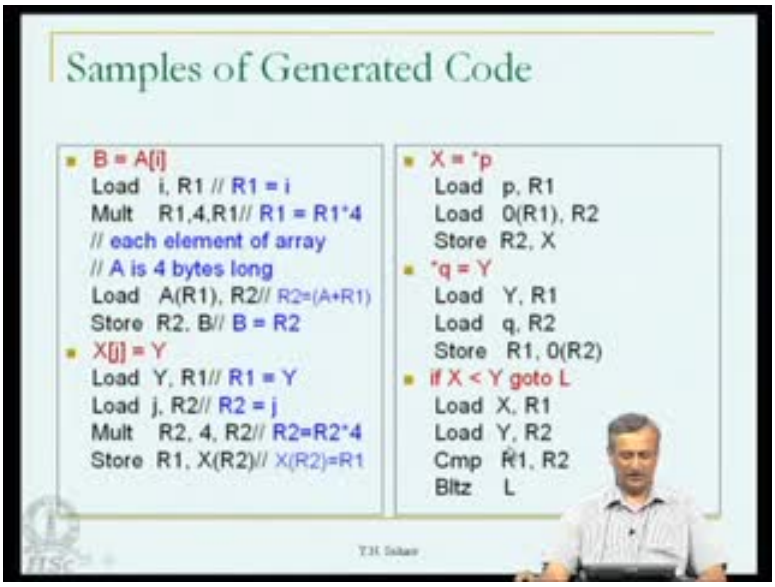
So, optimal assignment of registers to variables is very difficult to achieve. There are heuristics, which are used to achieve this with efficacy. Other issue is should we optimize for memory, time or save power? Each of these is actually very important in a

different context. For example, for embedded systems of one kind, it may be necessary to save memory and so we must optimize for memory.

In another embedded system, such as sensor network, the programs, which run actually save power because they all run on batteries. In our servers, it is necessary to optimize the programs to save time. So, they must run as fast as possible. Power and memory are not considerations here. So, the code generation strategy for any type of optimization is going to be different. The cost of the instruction sequence is abstract and so it can be memory space, time or power. We will see how this is done.

Is the code generator easily re targetable to other machines is a very important problem. Writing code generators is a difficult task. It is no easy task. There are thousands of details, which need to be kept track of. It is necessary and therefore that code generators are produced either automatically or parts of code generators are used in writing. Code generators for other machines can be produced automatically from the specification of the machine and we will see later. It is possible to use contexture grammars for the specification of the machine instructions. There is going to be a code generator, which is very similar to yak. Yak accepts contexture grammars and gives you a forward. This type of code generator accepts such specifications, generates code and produce code generators as output.

(Refer Slide Time: 52:22)



Let us look at some samples of generated code to understand what the issues are. We have shown several types of quadruples. B equal to A[i] is actually loading B from an array. X[j] equal to Y is assigning a value to an element of an array. X equal to star p assigns X, a value using a pointer.

Star q equal to Y assigns a value to allocation pointed by q. Here is a branch statement and let us see what is the code that is generated for each of these. B equal to A[i] and load i into R1. Now, the value of i is available. What is i? It is the offset within the array. A multiply R1 by 4 and then put the value in R1 itself. Why each of the elements of the array is assumed to be 4 bytes long? So, if you look at the byte version of the array A, each value is stored in blocks of 4 bytes. So, to get to the ith location of the abstract array A, we actually have to pass 4 star i locations and so that is what is done by multiplication here.

So, there are 2 instructions for this particular thing. Finally, load A of R1 with R2, sorry the other way load R2 with A of R1, R2 equal to A plus R1 and so contents of A plus R1. So, what we really did was? We take the index, which is R1. Go into array A and get the contents of that particular location. Now, it is A of i, put that into R2 and store the value of R2 in B. So, these 4 instructions are required to satisfy this 1 quadruple instruction. Similarly, X[j] equal to Y is simple and we have load Y with R1. So, load Y into R1, then load j into R2. Now, we need to compute the address of X[j]. So, multiply R2 with 4 and place it in R2. Now, we have gone to the place in X, where we need to store the value of Y. So, R1 contains Y and so store R1 into X of R2. This will store the value into X of j. So X equal to star p is simple, load p into R1. Now, load 0(R1),R2 and it takes the contents of R1 as an address and goes to that location to fetch the value in that particular location and puts it into R2.

So, an indirection here, store R2 into X puts this entire value, which is R2 into X. Star q equal to Y is also very similar. Load Y into R1, load q into R2, store R1 into 0 of R2. This is actually indirection. If X less than Y, goto L and load X into R1, load Y into R2 and now compare R1 and R2; if branch is less than 0, then the label L. So, these are some of the examples to show what kind of core that we need to generate. So, in the next lecture, we are going to look at the samples of static allocation, dynamic allocation. Then see how to generate such code using a simple methods and also with optimal methods. Thank you.