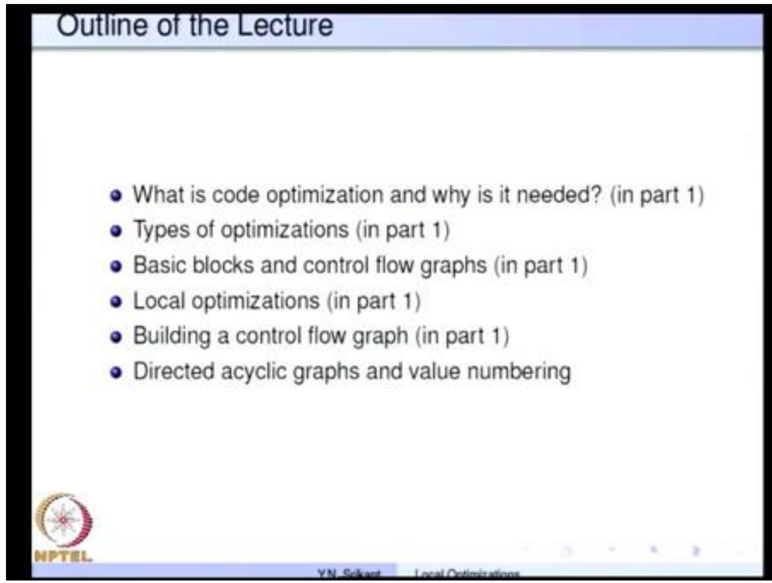
Principles of Compiler Design
Prof. Y.N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

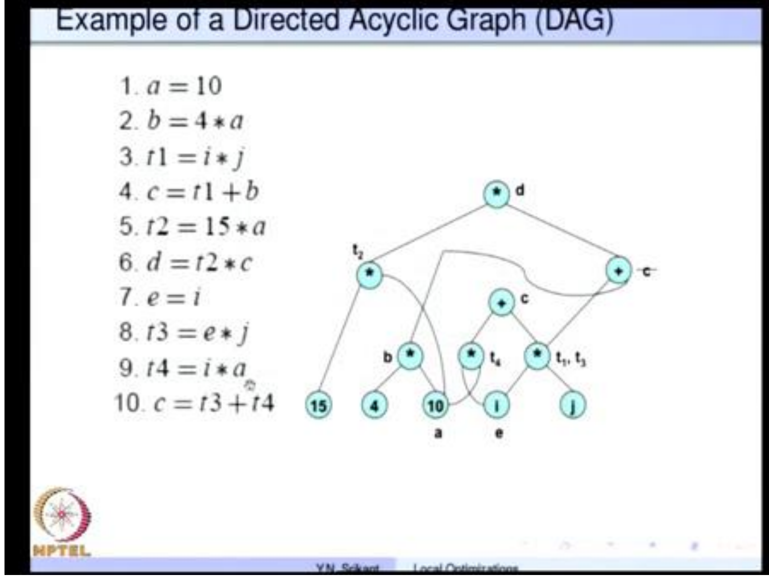
Lecture - 24
Control-Flow Graph and Local Optimizations Part-2
Machine code generation Part-1

(Refer Slide Time: 00:17)



So, in the last lecture we discussed the requirement for code optimization, types of optimizations, and we also discussed the procedure for building a control flow graph. Today we will continue with our discussion on value numbering and its use in local optimizations.

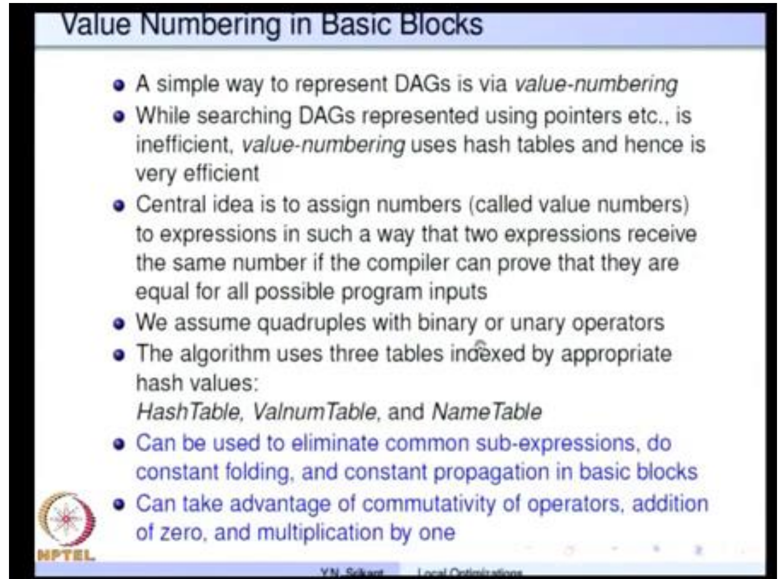
(Refer Slide Time: 00:40)



To do a bit of recap, so here is the example that I showed you in the last lecture. This is a basic block containing ten intermediate codes statements and this is the directed acyclic graph corresponding to this basic block. The most important feature of this directed acyclic graph is that it can enable several optimizations. For example, here is 4 star a which is actually a constant, because a is constant. So, this constant propagation and constant folding can be performed using the directed acyclic graph representation. It cannot be so easily performed using the quadruple representation.

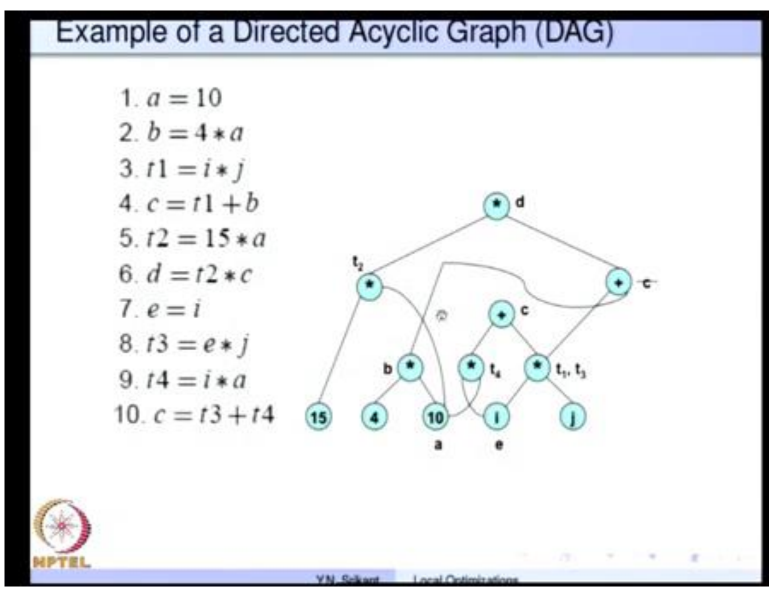
Here, is a you know here is an expression i star j and here is another expression e star j, but both are equivalent because e receives the value of i in during this execution. So, t 3 and t 1 are identical in value at all times and we can use t 1 instead of t 3 and delete quadruple number 8. So, this is called common sub expression elimination and this can also be performed using the directed acyclic graph representation.

(Refer Slide Time: 01:56)



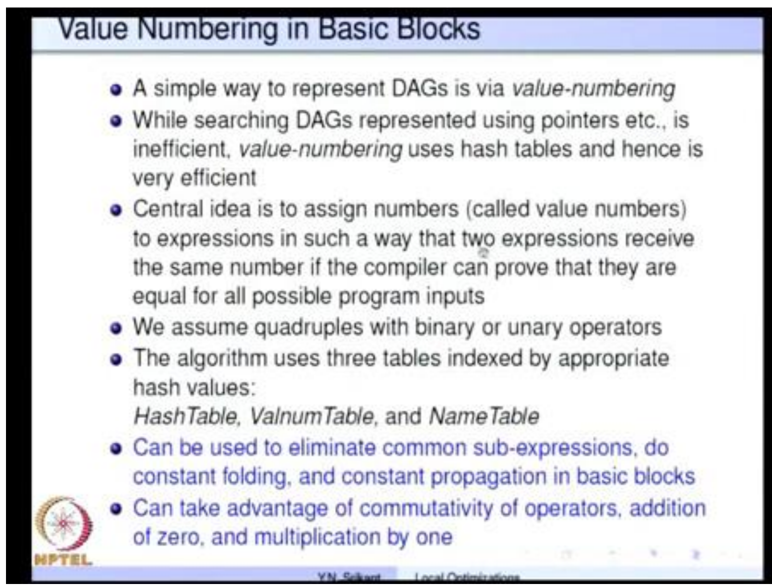
But as I explained in the last lecture it is you know useless to try and build the directed acyclic graph using link data structures.

(Refer Slide Time: 02:12)



Because, every time we want to locate some node you will have to start from top of the directed acyclic graph and search the entire graph as such so, this is a waste of time.

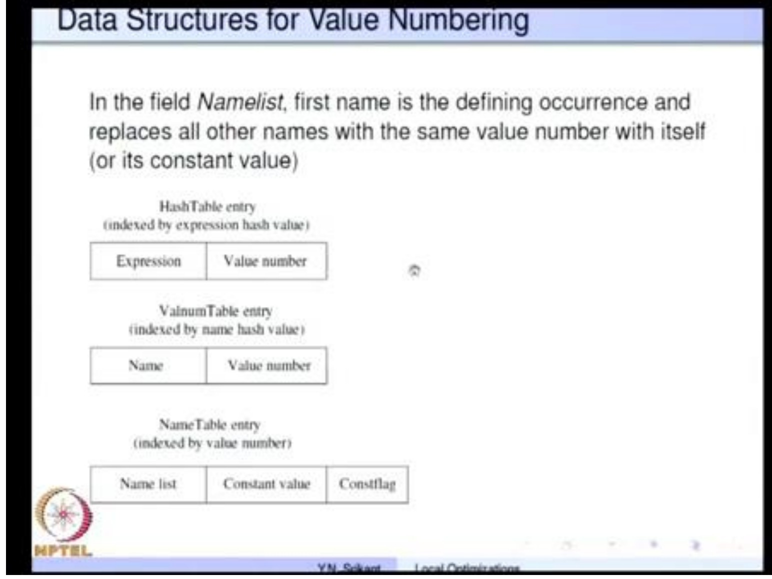
(Refer Slide Time: 02:21)



The technique that is normally used is called value numbering. So, what we really do is we assign numbers to expressions in such a way that two expressions receive the same number. If the compiler can prove that they are the equal for all possible program inputs. We use hash tables then you know hashing technique rather in this value numbering optimization. And we assume that there are quadruples with binary or unary operator.

So, there are 3 table hash table, value number table and name table. So, and the value numbering technique can be used to eliminate common sub expression do constant folding and constant propagation in basic blocks. We can also take advantage of the commutativity of the operators, addition of 0, multiplication of by 1 etcetera.

(Refer Slide Time: 03:22)



So, here are the 3 you know data structures that we are going to use in our technique. The first one is the hash table entry so the table will have records of this type, the first field in the record would be the expression itself and the second field would be the value number. So, what we do is you know compute a hash value for the expression using a suitable hashing function.

The hashing function must have you know must take it into consideration not only the two operands of the expression, but also the operator itself the type of the operator. And once we get a hash value we search the hash table and at that particular hash value we insert the expression and also a unique number called the value number itself. So, value number is nothing but, a unique number assigned to each expression. So, we are going to assign a value numbers using a counter.

The second table entry is the valnum table entry so this is the valnum table and the record structure is shown here. So, this is indexed by the hash value of the name so there is name and then the value number. So, the difference between these two is that this stores expressions and this store names. So, we cannot use the same hashing function for both expressions and names and that is the reason why we are using two different tables and two different hashing functions.

The third table is the name table its entry is shown here. So, the entry has one field known as the name list another field known as the constant value and the third field is

known as the constant flag. So, the this table is used to store the constant values of certain names so, if there is more than one name which actually stores the same value then it is formed in to a list. So, as I have written here the first name on the name list is the defining occurrence and it replaces all other names with same value number with itself or its constant value.

So, if there are 5 name here we really need to use only one of them in all places and the other 4 can be deleted from the program. If there is any computation associated with that name that can also be deleted along with it. So, if the name does not have a constant value associated with it then the cons flag will be false and the constant value field does not have any relevance. If it indeed has a constant value then this will be set to true and the constant value can be read from this field.

(Refer Slide Time: 06:18)

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$ (Instructions 5 and 8 can be deleted)

NPTEL

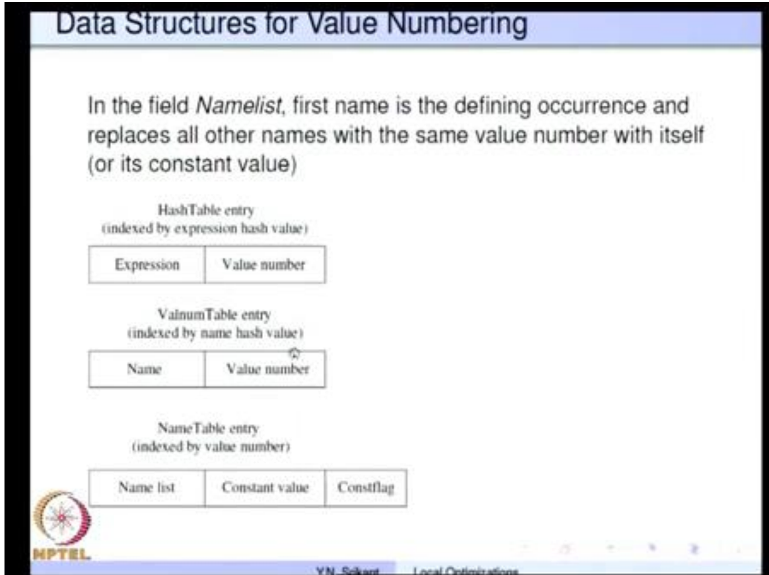
Y.N. Srikant Local Optimizations

So, let us understand the algorithm using a you know an example program. So, let us assume that this is the high level language program that is given to us so, 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 t I and then again c equal to e star j plus i star a. So, when we generate intermediate code you know for this sequence it obviously it will be split into many instructions. For example, we may have a equal to 10 then b equal to 4 star a as it is.

So, now the i star j goes into a temporary and then we have c equal to t 1 plus b, then again 15 star a goes into another temporary and d becomes t 2 star c. Then e equal to i

remains as it is t 3 is e star j, t 4 is i star a and c becomes t 3 plus t 4. So, we have these 10 quadruples in our basic block so let me explain how the value numbering technique is applied on this program. So, we start with the first quadruple so remember that there are three tables. The first one is the hash table for expressions, the second one is the name table for names.

(Refer Slide Time: 07:49)



The third one is rather the second one is the valnum table for names and third one is the name table for constants.

(Refer Slide Time: 07:53)

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$
		(Instructions 5 and 8 can be deleted)

NPTEL

So, this is a equal to 10 so the quadruple after value numbering remains the same what we do is enter this a into the valnum table and also the name table because this is a constant. And the effect of that is seen in the second quadruple itself as soon as we have 4 star a you know and of course, I forgot to mention one thing a is a new name that we have encounter. So, as soon as we enter it into the hash table we are going to generate a new value number for it and assign that particular value number.

So, when we have b equal to 4 star a the first thing we do is search for a in the valnum table it is indeed found there. And then you know by using its value number we can search the name table and find the value to be a constant that is 10. So, immediately we know that 4 star a can be computed at compile time itself you know it does not vary during execution. So, 4 star a can be computed as 40 and the quadruple can be rewritten as the b equal to 40.

Now, b is a new name so we generate a new value number for it enter it into the valnum table and assign it a new value number. So, this is the second thing that happens then we have the third one i star j. Similarly, i is you know i and j are new names and they need to be entered into the appropriate entries into the table, i star j is a new expression so we need a different value number for the expression also. So, the important thing is the name t 1 and the expression i star j will be assigned the same value number. So, whenever we find t 1 we know it is i star j and whenever we find a i star j we know that it is t 1 itself rather we can use t 1 for that i star j.

Now, this quadruple remains the same the next one is c equal to t 1 plus b so t 1 is already in the tables, b is also present in the tables we find that b is a constant it is from the name table. So, we can rewrite the quadruple as c equal to t 1 plus 40 and this new quadruple is now hashed into hashed and then entered into the hash table. The same thing happens with t 2 equal to 15 star a so, a is found as constant. So, it can 15 star a becomes 150 and the quadruple is rewritten, t 2 star c become as 150 star c, e equal to i remains the same.

The important thing here is e and i now get the same value number i had the same some value number before so, now e also gets the same value number. Now, when we search for e star j in the quadruple t 3 equal to e star j the value number of e is the same as the value number of i and value number of j already exist. So, in fact we end up hashing e

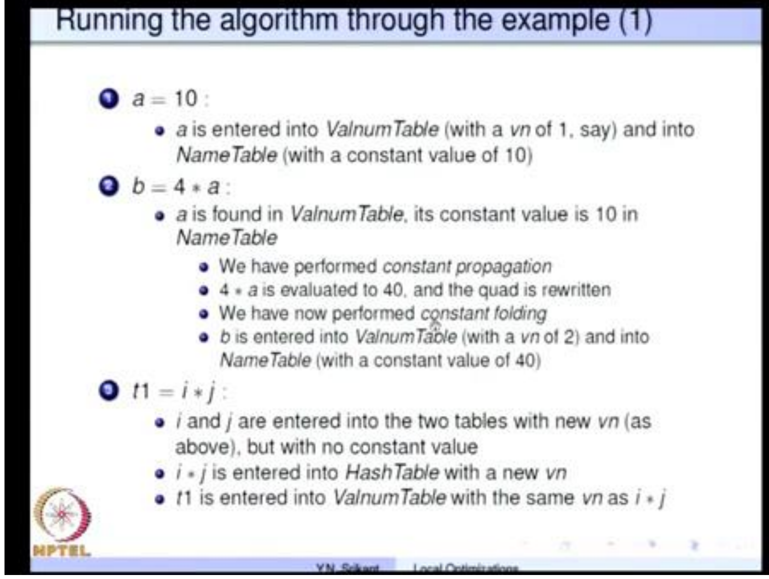
star j into the same slot as i star j because the value number of e is same as the value number of i.

That means, this particular you know once the value number is the same the expression remains same. So, essentially we can rewrite this t 3 quadruple as t 3 equal to i star j. In fact, even we can delete this quadruple because all occurrence of t 3 can be replace by t 1 you know because t 1 is also i star j then t 4 becomes i star a which is i star 10. And finally, we have another occurrence of c defining occurrence t 3 plus t 4. The old c is not relevant anymore so, this c is given a new value number and entered to the tables and instead of t 3 we are going to use t 1 and t 4 remains as it is.

So, 5 this particular t 2 equal to 150 is not necessary anymore because it is only use of t 2 was only in d equal to t 2 star c and once we have expanded t 2 to 150 this quadruple is unnecessary. This of course, is a common subexpression i star j has already being computed so, we do not have to compute i star j again and we can use t 1 place of t 3 we have already done that. So, quadruple number 8 can also be deleted so this is how we catch common subexpressions.

And this code is red code because we never use the value of t 2 again we can eliminate such red code and eliminate common subexpressions as well. And in the mean while we have all this has been possible because we propagated the value of a to these quadruples. it was prorogated to this place and this place. And of course, this place and here for 4 star a and 15 star a we also did constant folding that this evaluation of the expression at compile time.

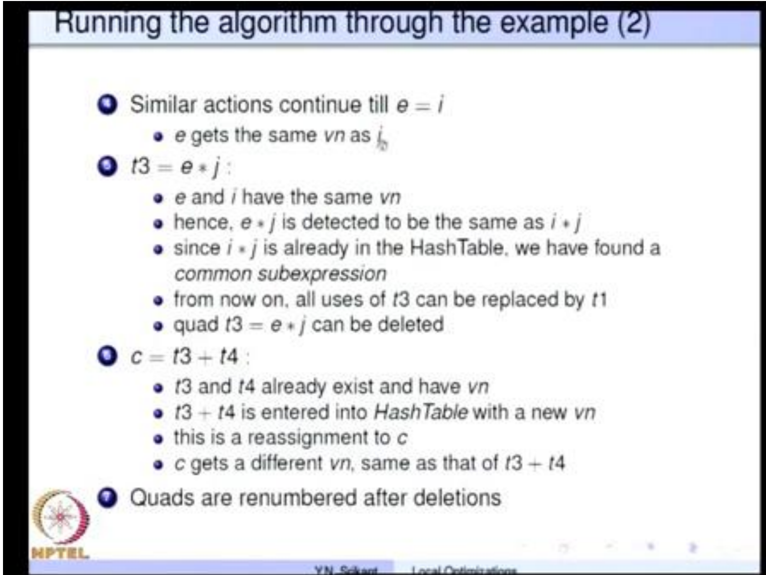
(Refer Slide Time: 13:26)



So, what I have written is the record of what I just now explained so a equal to 10. So, a is entered into the valnum table with a V n of say 1 and into the name table with a constant value of 10. Then b equal to 4 star a so, a is already found in the valnum table it is constant value is 10 in the name table. So, as I already explain we have performed constant propagation so we have evaluated 4 star a so that means, we have performed constant folding.

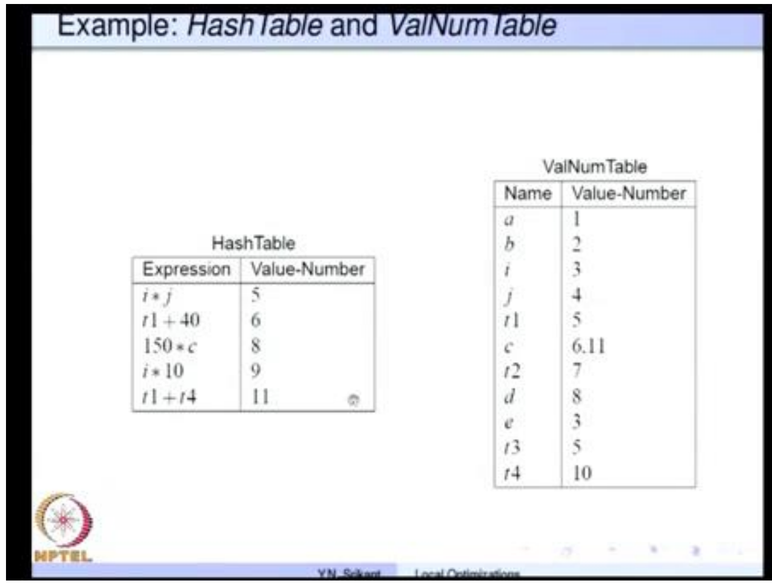
Now, b is enter in the valnum table now a new value number say 2 is given to it and into the name table with a constant value of forty, t 1 equal to i star j. So, as I already said i and j are entered into the tables i star j is entered into the hash tables and t 1 is entered into the valnum table with the same V n as i star j.

(Refer Slide Time: 14:24)



Similar, actions continue till e equal to i now e gets a same value number as i. So, in e star j, e you know we have e and i with the same value number so e star j is nothing but, i star j, i star j has already being entered into the tables so we have got a common subexpression when we search for e star j. So, from now on t 3 it can be replaced by t 1 and this can be deleted. So this of course, t 3 plus t 4 already exist and have new value numbers so t 3 plus t 4 is entered into the hash table. The assignment is to see so c gets a different value number and the old c should be killed so quadruples are e numbered after the deletions.

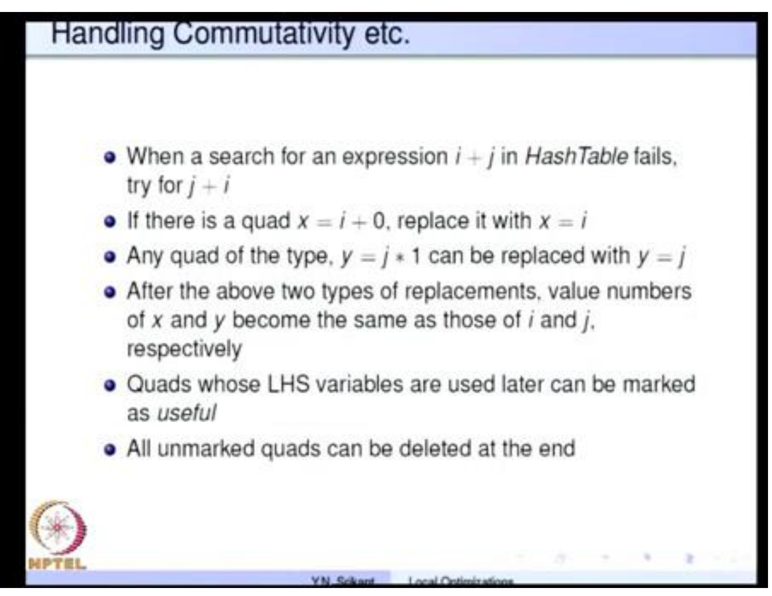
(Refer Slide Time: 15:12)



So, let me show you the tables after we have completed all this operations so i star j was entered into the hash table you know let us begin here. So, a got a value 1 b got a value 2 then we had i star j so a and j got the value of 3 and 4. Then i star j was entered into the table with a value of 5 so t 1 was assigned i star j so it gets a value of 5. Then we had c so it got a value of 6 which is the same as t 1 plus 40 then I will explain 11 very shortly.

So, t 2 gets a value 7 and this 150 star c gets 8 and d also gets the same value because it was an assignment. Then observe that e has the same value number as i so e has the same value as the 3 which is the i's value number, t 3 has the same value number as t 1 so that is 5 and finally, t 4 has the value number 10. So, i star 10 has the value number 9 and t 1 plus t 4 has the value number 11. So, this is how the value number table is used for the various optimizations.

(Refer Slide Time: 16:31)

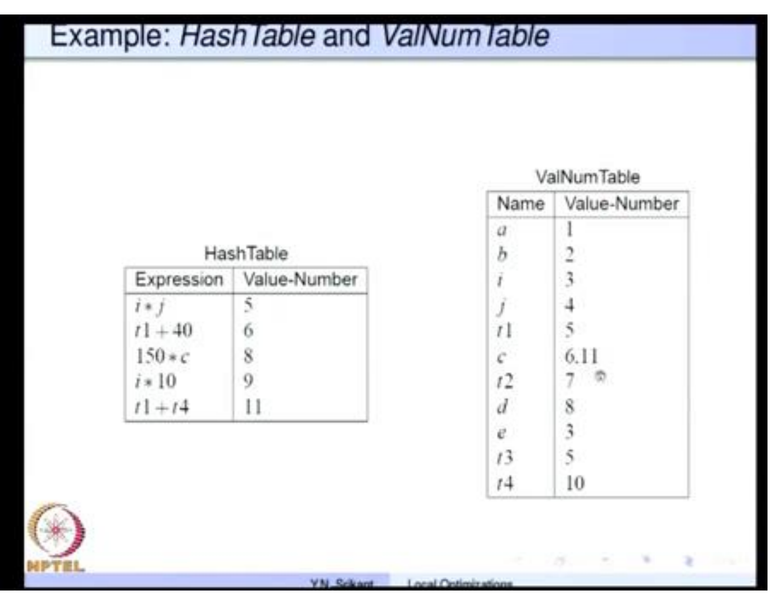


Now, so I mentioned that it is possible to exploit the commutatively of operators. So, in other words if you have an expression i plus j in the hash table or let us say j plus i in the hash table and we try searching for the other. So, we search for an expression i plus j it fails then try searching for the j plus i may be j plus i was already present in the table and the i plus j was not present. But, plus is commutative so whether we use i plus j or j plus i the value remains the same. And therefore, we can conclude that i plus j and j plus i are equivalent in all cases.

So, searching for one of the two forms and then using the value number of that particular expression, which was found in the hash table will not change in the value or rather result produced by the program. So, they can be deemed as the equivalent and given the same value number. So, we do not even have to enter it into the hash table so we because we always search both i plus j and j plus i in this hash table, if one of them is found we assume that there is an equivalence.

If there is a quadruple i plus 0 it can be replaced with the quadruple x equal to i. And similarly, the quadruple j star 1 can replaced with j so the assignment becomes y equal to j. So, in both these types of replacements the value number of x becomes the value number of y and value number of i becomes the value number j. So, quadruple whose left hand side variables are used later can be marked as useful all other unmarked quadrupled can be deleted at the end.

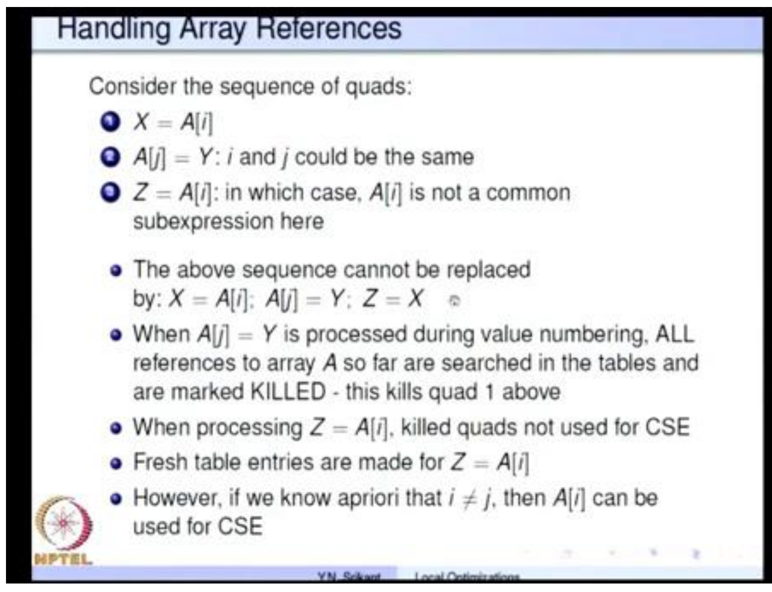
(Refer Slide Time: 18:28)



So, I also mentioned that c gets a new value number 11 because it was assigned again the old value of 6 is not used anymore. So, such value numbers you know the names are killed and then reused. So, the value number 6 should never be used again so it should be the value 11. So, that is how you know there could be the possibilities of i and j getting changed you know so and then we recompute the value of i star j. So, in such cases the second occurrence of i star j cannot be the same as first occurrence.

So, as soon as there is an a there is some you know i star i and j are changed so we need to make sure that we do not use the old value of a i star j. So, that is another thing that we need to keep in mind when we are using the value numbering technique. Of course, a sometime it happens automatically as soon as the value the variable i is resigned the value, the old value number of i is thrown away. So, we will never hash into the same location as the old value of hash value of i star j. So, that is in some sense automatically taken care of if we do this killing for the old value of c.

(Refer Slide Time: 19:53)



The next important thing is how to handle array references, scalar variables do not pose this problem, but consider the assignment of arrays. So, we have the first quadruple x equal to a i and then we have another quadruple following it a j equal to i. So, it is not necessary that these two are in the same you know immediately after one another. There could be other quadruples between 1 and 2 which have nothing to do with either x or a.

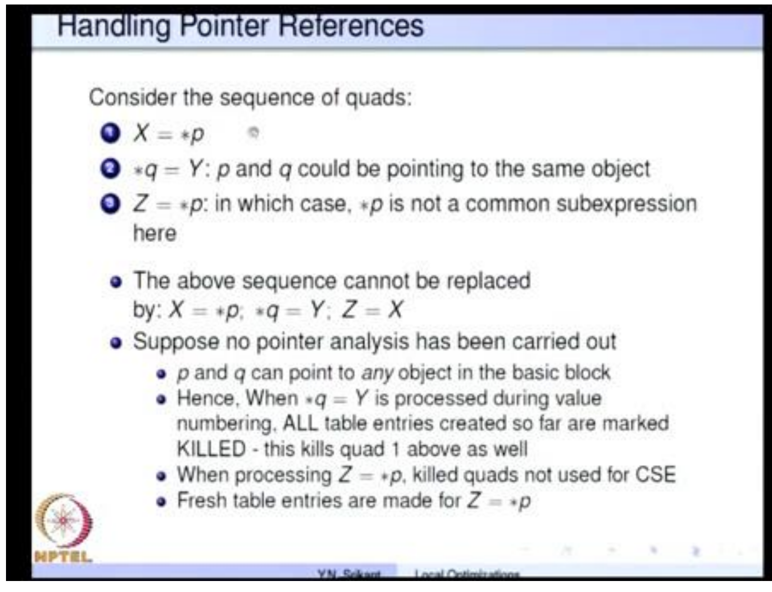
So, in that sense this is the next quadruple which is relevant to this first statement, a j equal to y suppose i and j are the same. So, then this quadruple really is a i equal to y that means, the old value of a i has changed. So, when we take another quadruple z equal to a i and suppose i and j were the same we really cannot use the old value of a i which is available in x. So, in other words we cannot replace the above sequence by x equal to a i, a j equal to y and z equal to x.

So, we are trying to do common sub expression elimination here by not using a i by rather using reusing the value of a i which is present in x, but this is illegal because most of the time we do not know whether i can be equal to j or not equal to j. If we definitely know that i and j cannot be the same then we can use this sequence, but if we have no idea whether i and j are equal or not equal then it is not possible to rewrite the sequence in this fashion.

So, the effect is when a j equal to y is processed during value numbering all the references to array a so far are searched in the tables and are marked as killed. So, there could be many such references in above this as soon as we get a j equal to y this is an assignment to a. So, there will be many references to a of i, a of k, a of l etcetera, etcetera above this a j equal to y. All these references will have to be killed they cannot be reused anymore because we do not know whether j equals any of those index values which have occurred earlier.

So, in this sense the x equal to a quadruple will also be killed, once it is killed the automatically z equal to a i will not replace it as will not be replace by z equal to x. So, as I already mentioned the fresh table entries have be made for z equal to a i. And unless we know i is not equal to j we cannot really do this common subexpression elimination.

(Refer Slide Time: 22:53)



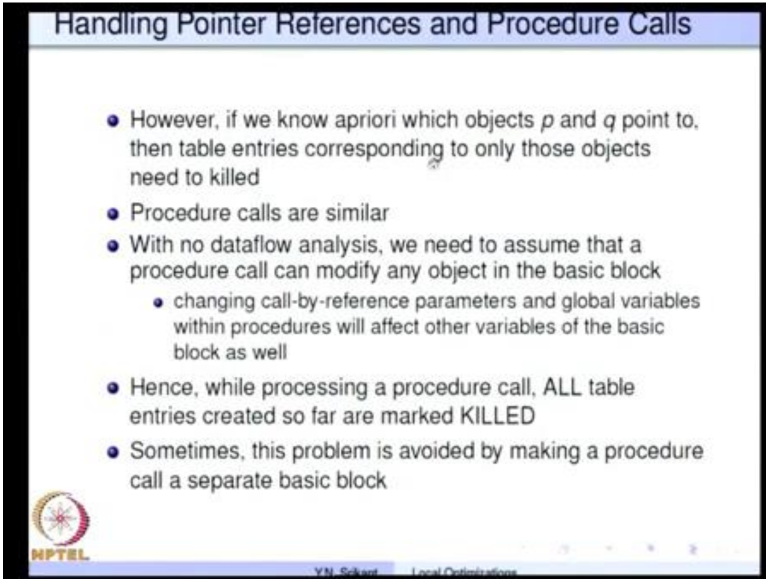
A similar, problem arises when we handle pointers and also procedure calls so, first let us look at pointer references. So, there is x equal to star p and then after a couple of

statements which do not affect x or p we have star q equal to y. So, suppose p and q could be pointing to the same object so in that case, we really in effect have star p equal to y which implies that the object p is pointing to has changed. Therefore, even though x stores the object pointed to by p we cannot say z equal to star p becomes z equal to y rather z equal to x.

So, in this sequence is illegal that is because in most cases we have no idea whether p and q are pointing to the same object or are not pointing to the same object. If pointer analysis has been carried out then the result of that pointer analysis may affirmatively say p and q do not point to the same object. So, in such a case this sequence can indeed be replaced by this sequence otherwise we cannot. So, if we do not know whether p and q point to the same object as soon as you process star q equal to y, all the table entries created so far will have to be killed.

So, p and q can point to any object in the basic block because we have no idea what p and q point to so we must assume that it can point to any object in the basic block. So, every one of the entries in the table up to this point will have to be marked as killed. That means, the table is automatically emptied and when we use when we process z equal to star p. The killed quadruples are not used for common subexpression elimination, fresh table entries have to be made for z equal to star p.

(Refer Slide Time: 25:06)



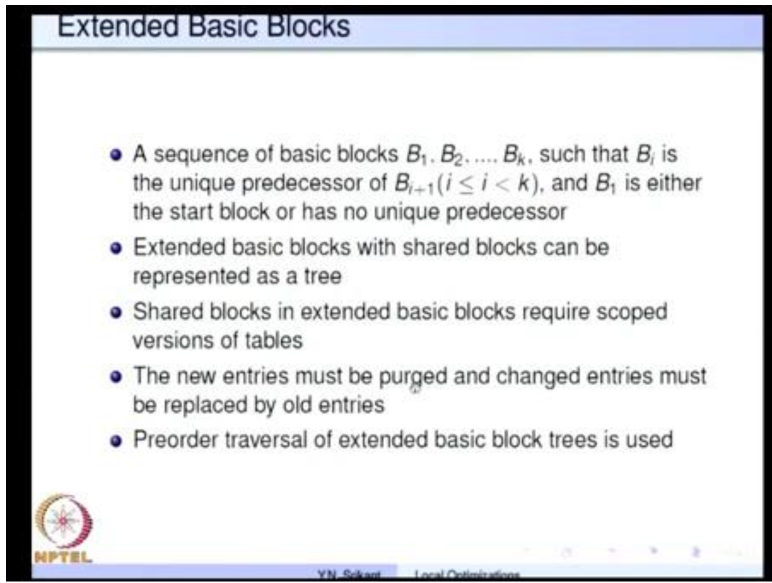
And of course, as I already said if we know that apriori which objects p and q point to. Then table entries corresponding to only those objects need to be killed we do not have to kill all the objects, all the statement and all the objects in the table. Procedure calls are handled in a similar way they also have similar side effects.

For example, if we did not analyze the program the analysis called dataflow analysis we need to assume, that a procedure call can modify any object in the basic block how can this happen. Suppose, we have a call by reference parameters and we change them call by reference parameters when they are changed the originals also get changed. So, the variables which are accessed in the basic block will also or anywhere else you know with the same name as the call by reference parameter will get changed.

Similarly, if there are global variables within procedures and the global variables are changed within the procedures. They will also affect the other variables of the basic block because global variables may be used anywhere in the program they will also be used in the basic block. So, all these will affect the variables in the basic block, why are we looking at only the basic block. The reason is the value numbering technique restricts itself to the basic block so it is affect will not percolate to other basic blocks.

So, we do not have to worry about we after not detecting common subexpressions across basic blocks at this time. So, we do not really have to worry about the effect of all these beyond the basic block, but if the procedure call can be separated into a different basic block. Then this problem automatically gets eliminated because we do not consider effects across basic blocks. If that is not so, if the procedure call is in the mid stuff a basic block then all the table entries created so far for that basic block will have to be marked as killed. And no common subexpressions can be detected and used. Therefore, usually we make a procedure call into separate basic block along with its parameter valuation and so on.

(Refer Slide Time: 27:36)

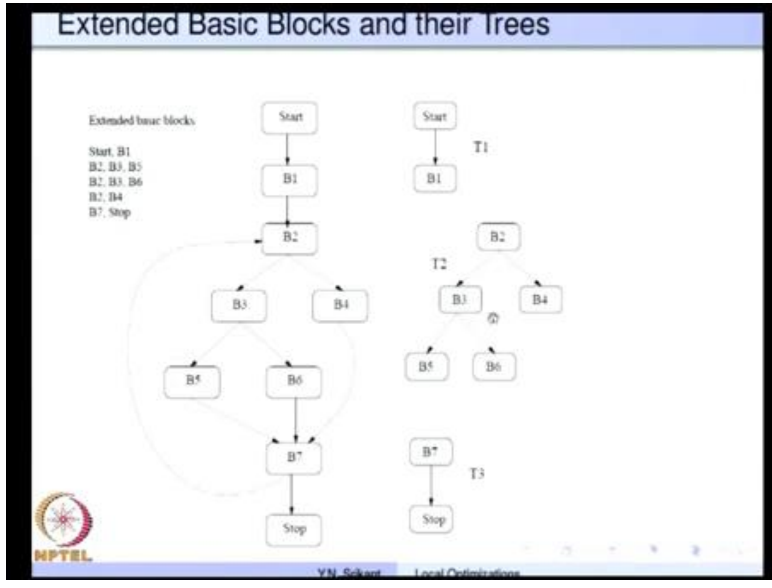


Now, having done this local optimization on basic blocks so is it possible to actually processes a sequence of basic block. And say look let me do you know common subexpression elimination value numbering etcetera on these extended blocks. It is indeed possible, but before that we will have to define what we mean by an extended basic block.

A sequence of basic blocks b 1, b 2 etcetera are b k such that b i is the unique predecessor of b i plus 1. So, if we have b 1 and b 2, b 1 must be the unique predecessor of b 2, b 2 must be the unique predecessor of b 3 etcetera, etcetera and b 1 is either the start block or has no unique predecessor. So, then you know it must be the start block or it should have no unique predecessor so we will see what that really means.

Extended basic blocks with shared basic blocks can be represented as a tree, I will show you an example of doing this as well. And shared blocks in extended blocks require scoped versions of the hash table, name tables and valnum tables. The new entries must be purged and changed entries must be replaced by old entries. So, we use a preorder traversal of extended basic block in order to perform the value numbering. So, now let me give you an example of extended basic blocks and then continue.

(Refer Slide Time: 29:15)



Here, is a flow graph so there are many basic blocks start then b 1 b 2 b 3 b 4 b 5 b 6 b 7 b 8. So, let us form extended basic blocks from this particular flow graph. So, we start at the first basic block and then we go to the next basic block, which it leads to which is b 1 and now start and b 1 are indeed in the same extended basic block. Now, let us try going to b 2 unfortunately b 2 does not have a unique predecessor in b 1 it also has another predecessor which is b 7. So therefore, we cannot include b 2 into the same extended basic blocks as start and b 1.

So, that is why we have separated start and b 1 into a different you know extended basic block. Now, we again begin at b 2 we go to b 3, b 3 has a unique predecessor b 2 then we go to b 5, b 5 has a unique predecessor b 3 we go to b 7, b 7 unfortunately has other predecessors as well. So, our search for extended basic block must blocks which can be included in the extended basic block stops here. So, we have b 2 b 3 and b 5 as another extended basic block so it is listed here.

So, if we take the other path we have b 2 then b 3 then b 6 again it has a unique predecessor b 3, but we cannot go to b 7 for the same reason that it has many predecessors. So, b 2 b 3 and b 6 will be another extended basic block then we have b 2 b 4 and we cannot include b 7 because of the same reason that it has many predecessors. So, b 2 and b 4 will form the will form a separate extended basic block finally, b 7 and stop will form the last extended basic block. So, we have 5 such extended basic block out

of which start comma b 1, b 7 comma stop do not share blocks with any others. So, these two are separated as two different extended basic blocks now the other three share blocks.

For example, b 2 is shared and then b 3 is shared so we can actually represent these you know these extended basic blocks using a tree structure. So, b 2 b 3 b 5 is 1 extended block, b 2 b 3 and b 6 is another extended block so they share not only b 3 they also share b 2. And b 2 is shared between these two extended basic blocks and also this b 2 b 4 extended basic block. So, these this forest of trees represents the set of extended basic blocks that can be formed out of this flow graph.

How do we apply you know value numbering on such trees that is our question. Applying value numbering on this tree which consist of start and b 1 is straightforward. We really can merge all the information in rather quadruples in b 1 with start and apply hashing. The same applies to b 7 stop as well, but the middle one is a little more complicated. We must perform a preorder traversals starting from the root of the tree so we go to we start at b 2. Now, the old value numbering technique can be applied to the contents of b 2 absolutely no problem there because it is just one single basic block, after the processing of b 2 is complete we go to b 3.

Now, because b 3 has a unique predecessor b 2 there is nothing wrong in detecting common subexpressions using an information and tables available in from b 2. So, b 2 and b 3 in some sense can be combined together to give you more benefit. So, we apply we just extend the hash tables and other tables of b 2 by including the entries of b 3 as well, but a word of caution here, we may want to undo this effect a little later as I will explain. So, we must keep these entries separated from the entries of b 2.

So, b 2 and b 3 together now you know detect common subexpressions so there may be something here which is reused here, so we do not have to keep that quadruple anymore. So, once the processing of b 3 is completed we go to b 5. The same argument holds here as well, the quadruples of b 2 and b 3 the tables of b 2 and b 3 can be reused to give advantage in b 5 as well.

So, expressions which are available in b 2 and b 3 can be reused here. So, common sub expression detection becomes even more effective. So, we extend the tables of b 2 and b 3 with the entries of b 5, but I already mentioned that we need to keep them separate.

Even though virtually a logically they are connected to the tables of b 2 and b 3 we will have to mark them as new entries. The reason is once the processing of b 5 is completed we go back to b 3 and the preorder traversal now takes us to b 6.

So, that means the effect of b 5 will have to now be undone, the reason is the effect of b 2 and b 3 can be seen in b 6. The subexpression of b 2 and b 3 can be still reused in b 6, but the tables of b 5 are not useful in processing b 6. Obviously, the control in the control flow graph will either go from b 2 to b 3 to b 5 or go from b 2 to b 3 to b 6 it will never go from b 5 to b 6.

And therefore, the tables of b 5 will have to be thrown away the tables of b 2 and b 3 alone will have to be kept. And if anything has been marked as killed in b 2 and b 3 that effect will have to be undone and the tables, which were present just before the entry to b 5 took place will have to be restored. Now, we can process b 2 b 3 and the entries of b 6 can take advantage of these. So, we do that and then we the preorder traversal goes back winds goes to b 2 and then it will visit b 4.

Now, it is time to undo the effects of not only b 6, but the effect of b 3 as well. So, the entries of b 3 and b 6 will have to be thrown away, the changes which were done to b 2 because of the processing of b 3 and b 6 will have to be undone. The old table which was present just before the entry to b 3 took place will have to be restored. And then we enter b 4 process b 4 completely taking advantage of the entries of b 2. So, this is what I meant here when we said the shared blocks in extended basic blocks require scoped versions of tables. So, these are very similar to the scoped versions of symbol tables which we used when we processed blocks nested blocks and nested procedures.

(Refer Slide Time: 37:32)

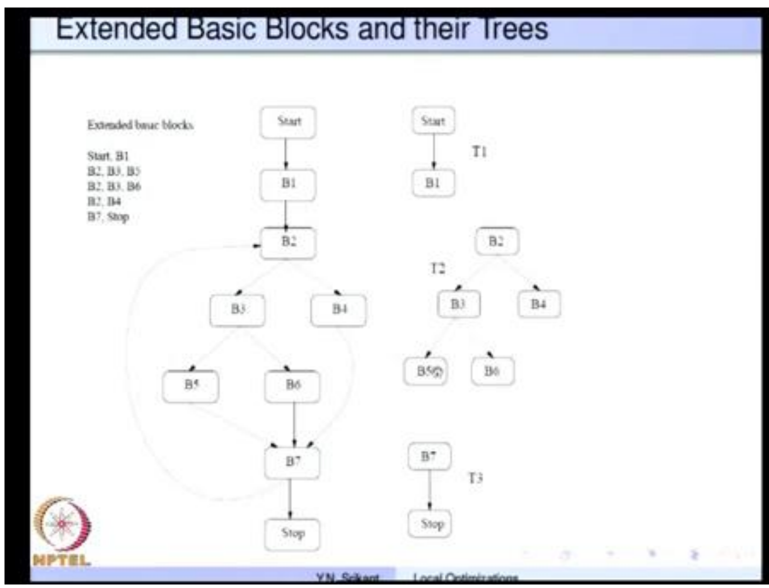
Value Numbering with Extended Basic Blocks

```
function visit-ebb-tree(e) // e is a node in the tree
begin
   // From now on, the new names will be entered with a new scope into the tables.
   // When searching the tables, we always search beginning with the current scope
   // and move to enclosing scopes. This is similar to the processing involved with
   // symbol tables for lexically scoped languages
   value-number(e.B);
   // Process the block e.B using the basic block version of the algorithm
   if (e.left ≠ null) then visit-ebb-tree(e.left);
   if (e.right ≠ null) then visit-ebb-tree(e.right);
   remove entries for the new scope from all the tables
   and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
   for each tree t do visit-ebb-tree(t);
   // t is a tree representing an extended basic block
end
```

So, now the function visit ebb tree for you know does the value numbering on these extended basic block trees so, it is quite simple. So, whenever we enter the, enter a particular node and function visit ebb tree is called on that node. The new names will be entered with a new scope into the tables, when searching the tables we always search beginning with the current scope and move to the enclosing scope.

(Refer Slide Time: 38:04)



In other words, when we are processing b 5 we must search the tables for b 5 first and then the tables for b 3 and then the tables for b 2.

(Refer Slide Time: 38:14)

Value Numbering with Extended Basic Blocks

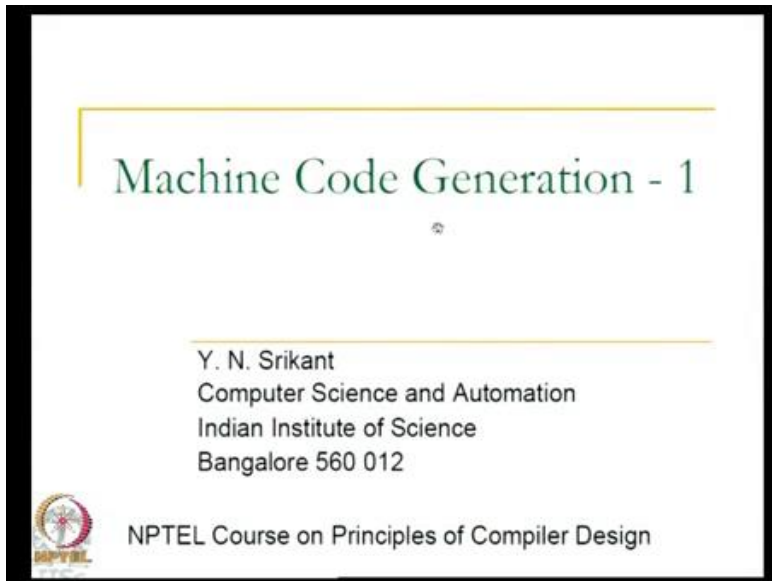
```
function visit-ebb-tree(e) // e is a node in the tree
begin
  // From now on, the new names will be entered with a new scope into the tables.
  // When searching the tables, we always search beginning with the current scope
  // and move to enclosing scopes. This is similar to the processing involved with
  // symbol tables for lexically scoped languages
  value-number(e.B);
  // Process the block e.B using the basic block version of the algorithm
  if (e.left ≠ null) then visit-ebb-tree(e.left);
  if (e.right ≠ null) then visit-ebb-tree(e.right);
  remove entries for the new scope from all the tables
  and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
  for each tree t do visit-ebb-tree(t);
  // t is a tree representing an extended basic block
end
```

So, this is very similar to the processing involved in symbol tables for lexically scoped languages. Now, so we call value number e dot b so process the block e dot b using the basic block version of the algorithm. So, let us assume that this does that now this is these two are nothing but the preorder traversal lines. If e dot left not equal to null then visit ebb tree on e dot left and then the right one. If e dot right equal to null visit ebb dot tree e dot right.

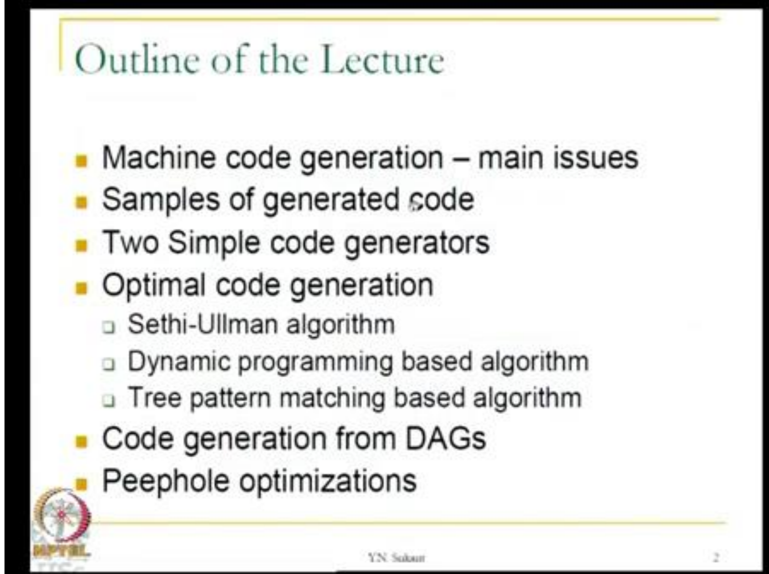
So, remove the after these two left and right sub trees have been visited we remove the entries for the new scope from all the tables and undo the changes in the tables of enclosing scopes. So, the main calling routine simply says for each tree t do visit ebb tree t. So, we just do this for the all the trees in that forest and that completes the value numbering. So, this brings us to the end of the lecture on local optimizations.

(Refer Slide Time: 39:31)



So, welcome to the lecture on the machine code generation.

(Refer Slide Time: 39:40)



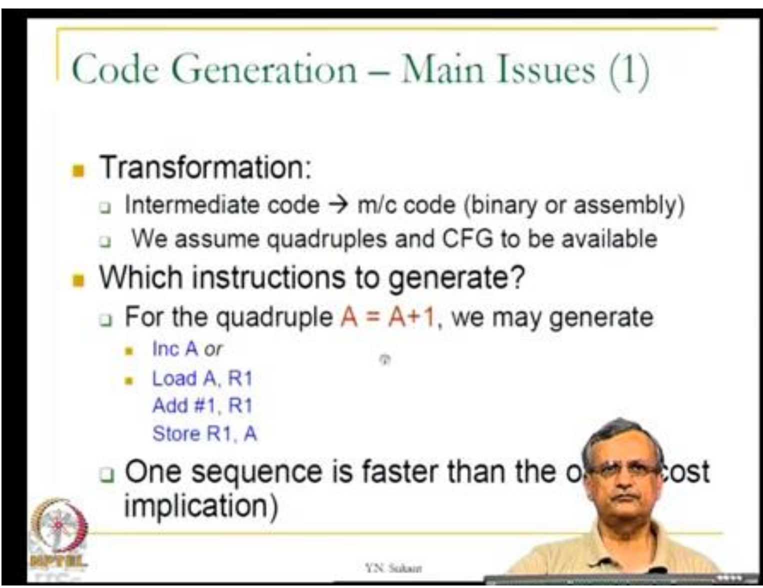
So, far in the previous lectures we have studied you know lexical analysis, syntax analysis, intermediate code generation, semantic analysis. And then we also saw some of the simple optimizations which can be local optimizations, which can be done on the intermediate code. Now, it is time to understand how machine code can be generated from the intermediate code. And after this long session on a machine code generation we will move on to the machine independent optimizations and so on and so forth.

So, in this lecture will first understand the main issues in machine code generation, we will look at few samples of generated code. Consider, very simple code generators two of them to be very precise. And then we consider the very important topic called optical code generation, which in you know in fact generates a best possible code. We study three types of optimal code generators, the first one is the classical code generation algorithm due to Sethi and Ullman dating back to 1970.

The second one is the dynamic programming based algorithm again this is this dates back to 1976 or so, but it is still in used today. The third is the tree pattern matching based algorithm for machine code generation, which uses a dynamic programming and also tree patterns. And after this we will understand how to generate code from directed acyclic graph.

The reason is we definitely you know do local optimizations using directed acyclic graphs. So, what we get after the optimization of the basic block is still a DAG. So, we must know how to generate a machine code from DAGs and then we will look at a special class of optimizations called Peephole optimizations, which are carried out on the machine code.

(Refer Slide Time: 42:04)



So, here what exactly is machine code generation, it is a transformation, it is a transformation from intermediate code to machine code. So, when we say machine code, machine code could be in binary form or it could be in assembly form. This really does

not matter to us because both assembly code and binary code are equally useful. It is just that if we generate assembly code we need to use an assembler to transform the assembly code to binary.

Whereas, if we want to generate binary code directly then we may end of doing all the work of the assembler itself. So, in our lectures we will assume that we are generating assembly code and not binary. Of course, as usual we will assume quadruples and the control flow graph to be available to us so, that we can processes them to generate code. So, that is the transformation we talk about intermediate code to machine code. So, in this transformation which instructions should we generate?

The problem is for some of the quadruples it is you know definitely correct to have more than one sequence of machine code in this transformation. For example, take a simple increment a equal to a plus 1, it increments the value of location a right. So, for this quadruple we may actually generate a single machine instruction called increment a, if such an instruction exist in the machine. In some machines increment operation cannot be carried out on a memory it can be carried out only on registers. So, in some other machines there may be no instruction for increment at all.

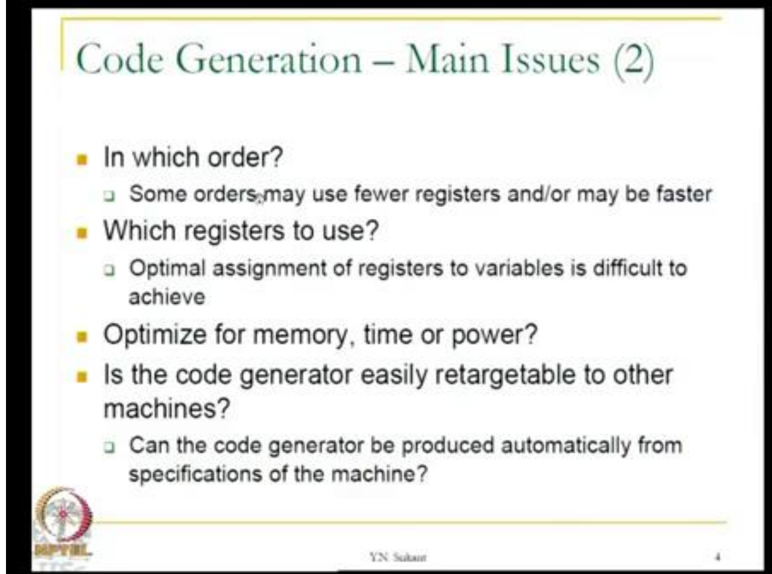
So, in such machines we may want to generate load a comma r 1, this is the risk architecture style, then either add 1 to r 1 are increment r 1 then store r 1 to a. The reason we had to do this was the machine performed all the arithmetic only on registers, the only operations from and to memory for the load and store. So, we had to bring the operand into the rather the a which is nothing but the operand of the right hand side. We had to bring it into a register alter its value and put it back into the location.

So, one of these you know obviously this decision will be based on the machine. So, this is one to many mapping, which mapping is really good for us will have to be chosen by the code generator. One sequence actually may be faster than the other sequence in certain cases. In this case of course, increment a if you had both options we would want to generate increment a because this is just one instruction and much faster than executing three instructions.

So, the implication is there is a cost attached to the execution so, if we are able to take care or compute the cost of the instructions which you are generated. Then a rather may

be generated we may be able to make a decision regarding the sequence that is to be generated.

(Refer Slide Time: 45:58)



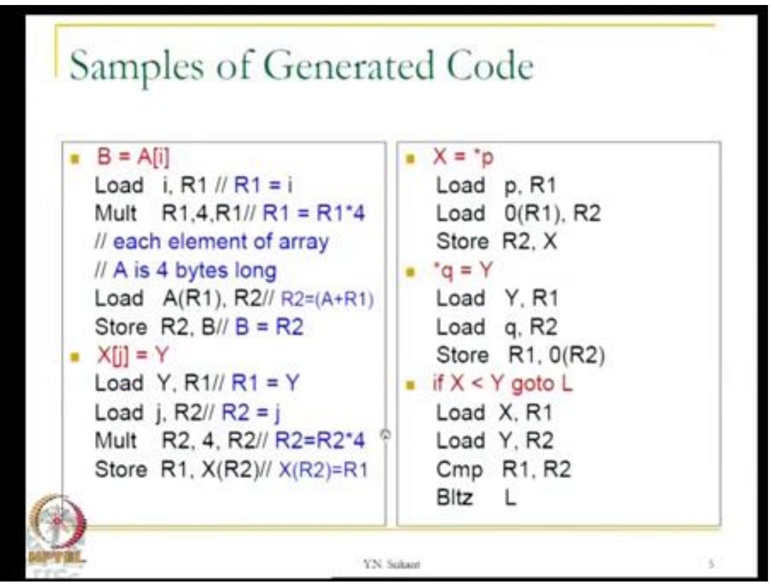
The next one, the next issue is in which order should the machine instructions be emitted. So, some orders may use fewer registers and or may be faster. So, in such case the again the cost implication comes into picture and the number of registers used etcetera comes into picture. The compiler would like to evaluate these possibilities and a do it is best. The next issue would be which registers are to be used in the machine code that is generated.

So, optimal assignment of registers to variables is very difficult to achieve. In fact, the problem is very famous it is called as the register allocation problem. So, this will say certain variables it will the allocation algorithm will decide on which variables should be placed in registers and which variables need not be placed in registers. So, optimal assignment of such registers is such you know registers to variables is a very hard problem n p complete. So, there are heuristics which can be used for that purpose we will study these heuristics at a later point in time, should be optimized for memory, time or power.

So, we already mentioned in the lecture on optimization that we can optimizes the space, the time or energy it power consume by the program, is the code generator easily retarget able to other machines. The problem is code generation is very tricky and very difficult.

So, if you are able to actually write a specification of the machine for which we want to generate code. And the code generator can be produced automatically from the specifications of the machine that is the best scenario. So, in such a case we simply have to you know have a code generator for which you know processes the different machine specifications. And then easily we can generate code generators for any type of machine. So, in the absence of that we will end of writing machine code generators for each one of the machines by hand.

(Refer Slide Time: 48:35)



So, let us look at some of the samples of generated code. To begin with have I listed actually the more complicated types of statements here, straight forward quadruples such as a equal to b plus c are very easy to handle. So, let us look at the more complicated one such as involving arrays and pointers and conditional statements and so on. So, this is a quadruple b equal to a i so the first the of course, this means we will have to take the i'th element of the base address a, from the base address a.

So, remember this a i does not mean it is a single dimensional array, we have already understood how to translate you know multidimensional array references to single dimensional memory reference. So, this a really can be the base address of the starting address of a multidimensional array. It need not be the starting address of a single dimensional array, but for our purpose it really does not matter at this point it is a sequence of bytes. And we simply want to take the you know appropriate byte from

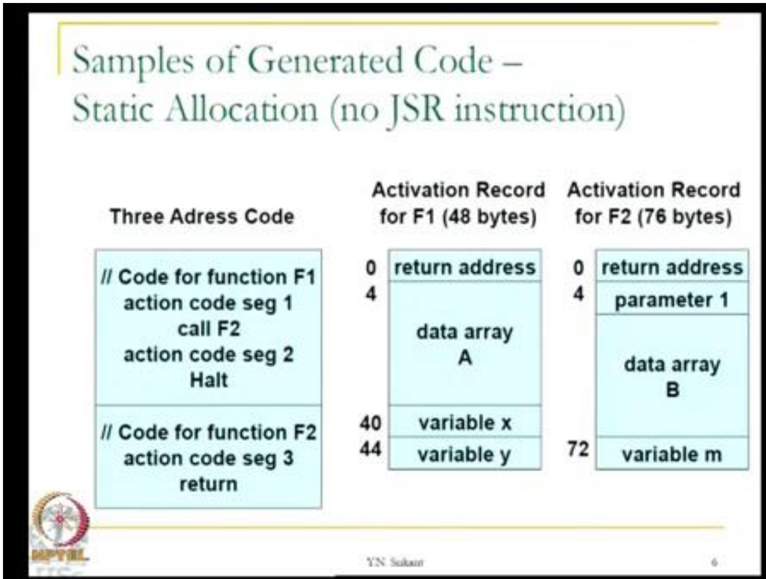
starting from the address a and appropriate 4 or 8 bytes from the starting address a and then put it into b.

Let us see how the machine code for this looks like so, we load i into the register r 1 and then we multiply the value of i by. The reason is we are assuming that every entry in the array a corresponds to four bytes so we say r equal r 1 equal to r 1 star 4. Then we use indexed mode of addressing so, starting from the base address a this is the index a of r 1 comma r 2. So, take the contents of a plus r 1 and put it into r 2 finally, store r 2 into b. So, we have to generate four instructions just for this one intermediate code.

The same is true for x j equal to y as well, we have load y comma r 1 so no problem load j comma r 2 so that is left hand side part. Then multiply r 2 by 4 so, that takes us to the appropriate place within the array where we can load y and then store r 1 comma x of r 2. So, x of r 2 gets r 1 so this is the indexed assignment. So, this is the sequence of instructions generated for just one intermediate code here.

Similarly, x equal to star p translates to load p comma r 1, p is a pointer an address. And then with a you know this is nothing but indirect addressing 0 r one implies address of contents of contents of r 1 goes to r 2 and we store r 2 to x, star q equal to y is similar. So, load y comma r 1 and load q r 2 and store r 1 comma 0 r 2 so, again this is the indirect mode of addressing. If you have if x less than y go to l then you know we load x 1 x into r 1, we load y into r 2 compare r 1 and r 2 and branch on less than 0 to l. So, this is the sequence which appropriately you know performs gives you the same effect as if x less than y go to l.

(Refer Slide Time: 52:27)



Now, let us look at a slightly more complicated scenario involving activation records static allocation, dynamic allocation, subroutine jump etcetera, etcetera. The three address code has say code for this is the code for function f 1 so, this is these are the two functions that we have. So, within the code for function f 1 we have some action code segment a number of instructions in intermediate code and then there is a procedure call.

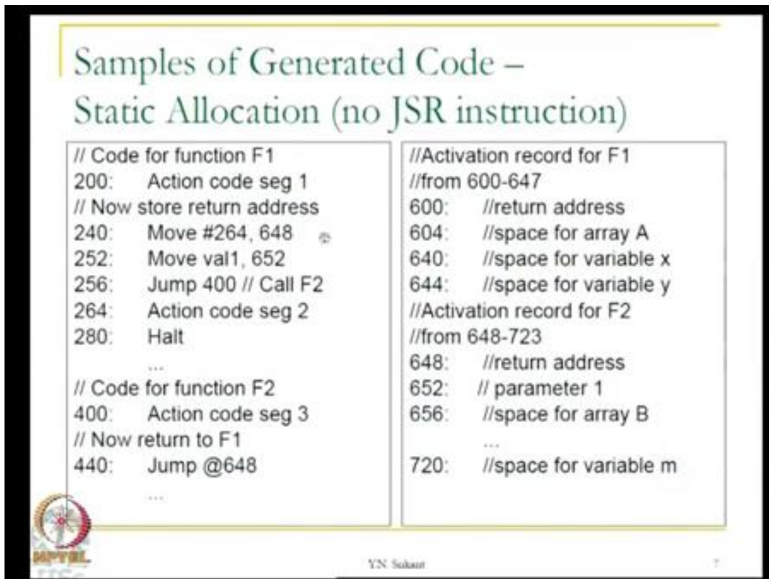
So, in this example we are concentrating on the procedure call and activation records etcetera the rest of the intermediate codes are not very important to us at this point. So, there is a call f 2 so we are calling this function f 2, action code segment 2 after the return we execute this piece and then halt, code for function 2 has an action code segment and then a simple return. So, we are looking at static allocation and we assume that there are no jump subroutine instructions in the machine instruction set so, we will have to actually do a jump.

So, because we have to do a jump we also need to store the return address in the activation record itself. So, the activation record is a static activation record in other words, it will never change. You know the same static the activation record is used for all invocations of the function f 1 and f 2 no recursion is possible which static allocation. So, this does not create a problem for us.

So, here is the first one in the offset 0 starting from the top we store the return address then we have a data array a we have the variable x and variable y. So, totally 48 bytes are

needed for this f 1. Similarly, for f 2 we require 76 bytes return address there is a parameter as well then another an array and a variable m.

(Refer Slide Time: 54:39)



So, here is the code that may be generated by a compiler code for function f 1 so, say we start from address 200 then there is the code for segment 1. Now, we it is time to call a the function so we have you know move hash 264 comma 648, 264 is nothing but the address of the return address where we need to return. Then the parameter is moved into the location 652, which is the place for the parameter. Then we have a jump this is the call to this you know this 400 is the function.

And then we have in this function when we want to return we take the return address from the stack and then jump at 648, 648 stores the return address. So, we come back to this action code segment for 2 execute it and then we halt. So, and here is the activation record format for f 1 600 to 647 and then the action activation record for f 2 648 to 723. So, we will stop the lecture here and continue in the next class.

Thank you.