Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

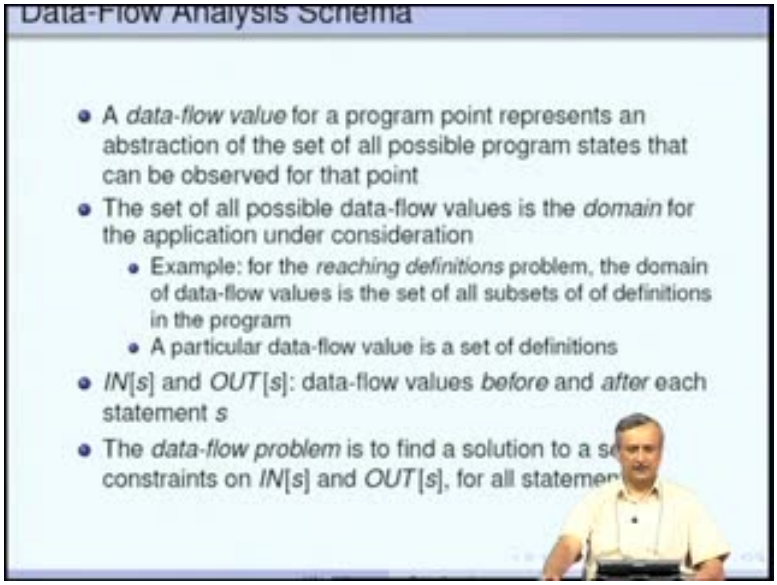
Module No. # 08

Lecture No. # 21

Data-flow Analysis- Part 3

Control Flow Analysis

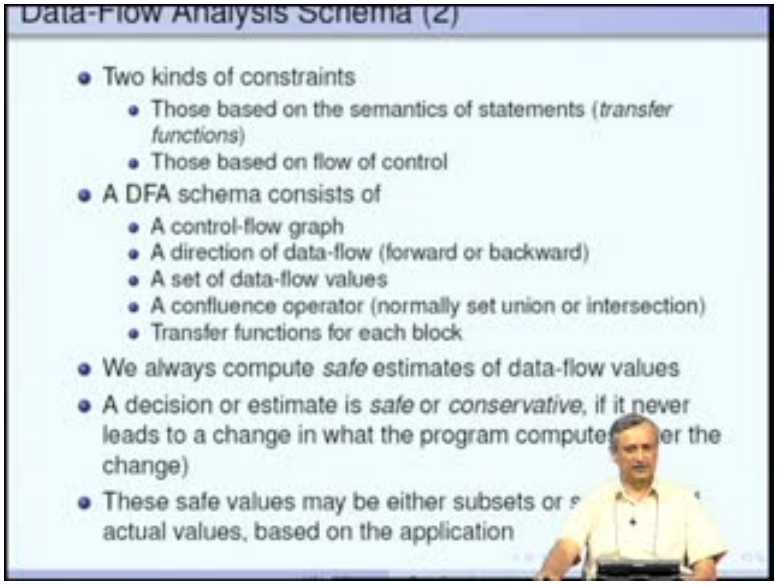
(Refer Slide Time: 00:22)



Welcome to part three of the lecture on dataflow analysis. Briefly reviewing what we did in the last two lectures, dataflow value for a program point represents an abstraction of the set of all program states that can be observed for that point. Then we studied the the reaching definitions, available expressions, problems in previous lectures.

In both the problems, we computed the in and out sets, which are the dataflow values before and after each statement. The general problem of dataflow analysis is to find a solution to the set of constraints on IN and s for all statements s.

(Refer Slide Time: 01:12)



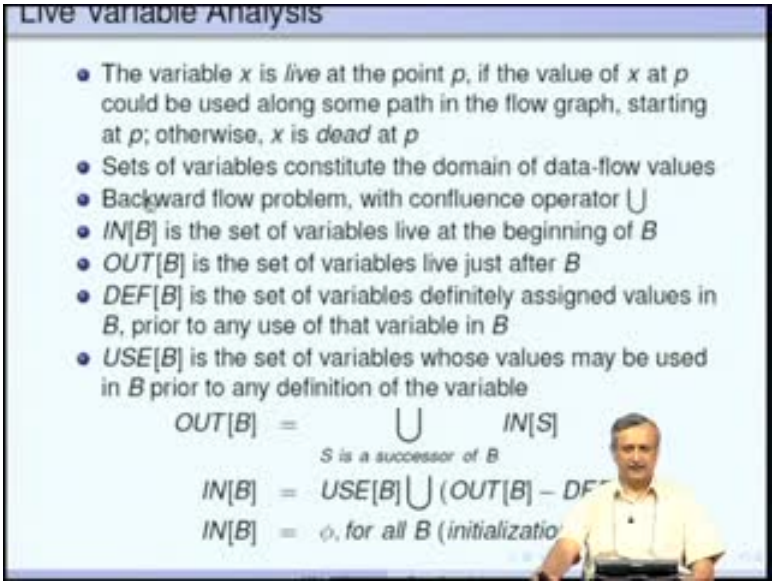
There is a dataflow schema that we have been describing for the dataflow analysis problems. So, the schema consists of a control flow graph, a direction of dataflow, either forward or backward, a set of all dataflow values, a confluence operator and some transfer functions for each block.

For the reaching definitions problem, you know the direction of data flow was forward. The set of data flow values or the domain of data flow values are the actually the sets of the reaching definitions. Each set of the reaching definitions was a data flow value.

The confluence operator was union for the reaching definitions problem. The transfer functions really described how 10d in change; so, that is precisely how we described the reaching definitions. Similarly, we described the available expressions problem also. A very important point is we always compute safe estimates of dataflow values. So, what are the safe estimates? That really depends on the application. So, it is a conservative estimate.

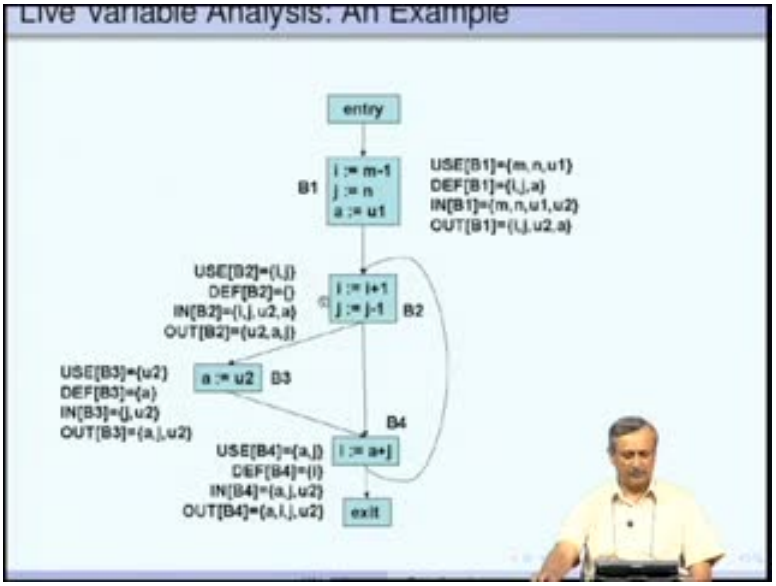
For example, in the reaching definitions case, we may include more definitions in the set of the reaching definitions, which says something may not reach, but we still want to include it, because we are not very sure. Whereas, for the available expressions, unless we are certain that the expression is available at a point, we are not going to include it in the set. So, in that process, we may actually lose some of the expressions, which may actually be available but a safe estimate is always like that.

(Refer Slide Time: 03:09)



We were discussing the live variable analysis problem. What is the live variable analysis problem? If we say that a variable x is live at a point p, then you know the value of that particular variable at that point is going to be used along some path in the flow graph, starting at p. If this does not happen, then we say that x is dead at p. So, the sets of variables constitute the domain of dataflow values here. Each dataflow value is a set of variables. This is actually a backward dataflow analysis problem and the confluence operator is union.

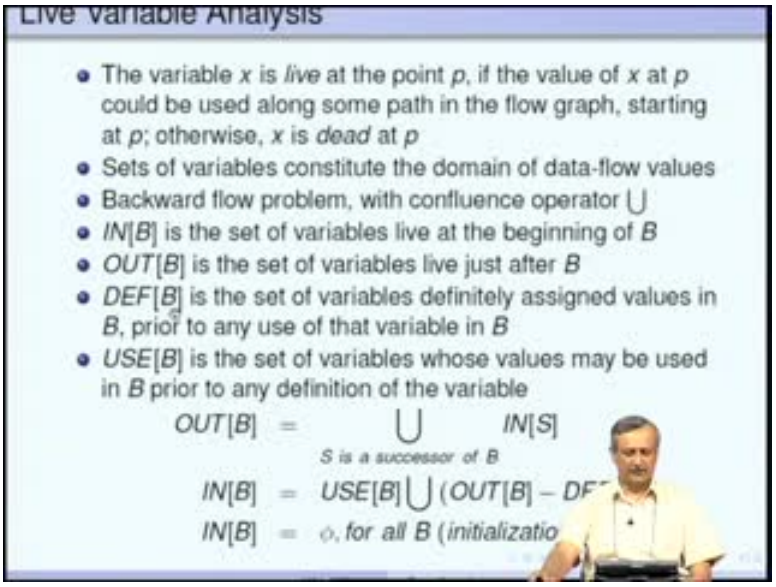
(Refer Slide Time: 04:07)



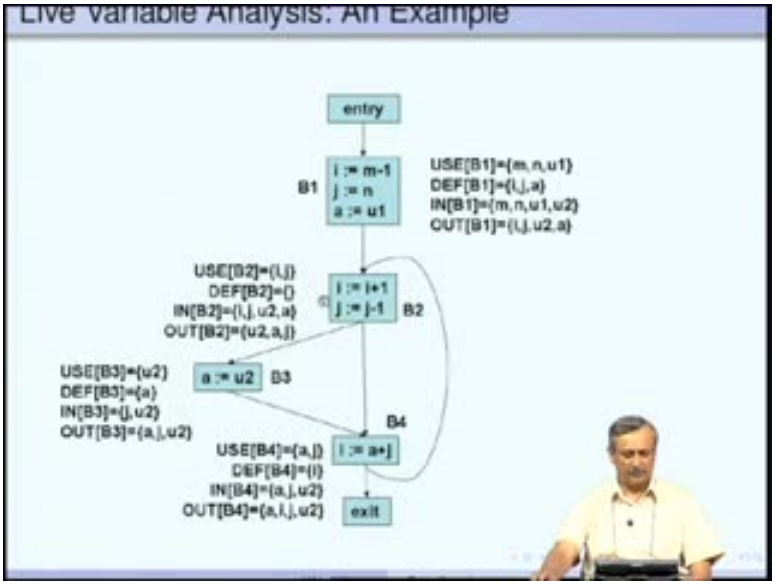
When we say it is a backward analysis problem, for example, if you look at the control flow graph here, it does not mean that the traversal of the flow graph is in reverse order, that is not the point. A forward or backward flow does not have anything to do with the traversal of the basic blocks in a particular order. In fact, I must add here that the order in which traverse the dataflow traverse the basic blocks of the control flow graph is immaterial.

The same solution will be obtained irrespective of the order in which basic blocks are traversed. It is just that there are some orders which give solution very quickly, whereas, some other orders may not give the solution very quickly. This is true for both forward analysis problems and also backward analysis problems.

(Refer Slide Time: 05:02)



(Refer Slide Time: 05:37)



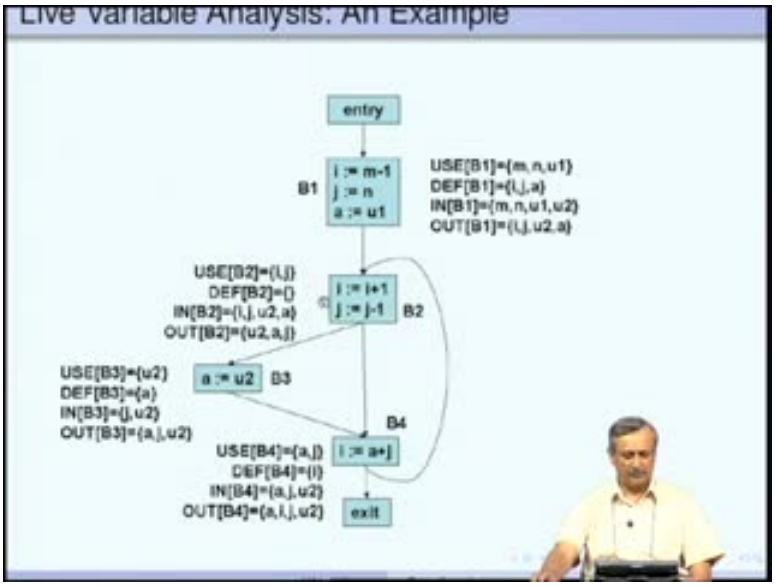
In the case of live variable analysis, IN B is the set of variables live at the beginning of B and OUT B is the set of variables live just after the basic block B. The gen and kill correspond to the 2 sets, USE and DEF. So, USE B is the gen set, it is the set of variables whose values may be used in B prior to any definition of the variable. In the example here, USE of B1 is m n and u1 – why, because m n and u1 are used on the right hand side of the three assignment statements before they are assigned any values. In fact, they are not assigned any values at all. Whereas, in the case of B2, i and j are used; before they are assigned a value. So, you must observe that even in the assignment statement this is true - i is accessed first and then assigned. In this case, it is just u2 and in B4 it is a and j.

DEF is the set of variables definitely assigned values in B prior to any use of that variable in B. i, j and a are assigned values and they are not at all used. So, DEF B1 contains i j and a, in the case of B2, there are no variables assigned values before they are utilized.

For example, this i is the old I, and this is the old j as I mentioned before. So, this set DEF of B2 is null, a is assigned here, and therefore, it is u2 a. i is assigned here, so, DEF of B4 is i. So, this is how the values are really computed. Only this is an exception - i and j are already used before they are assigned any value. So, DEF of B2 is 5.

The equations are here - OUT B is the union of the IN values of the successors of the basic block B, and in B is as usual, it has the gen part and then the kill part here. So, USE B union OUT B minus DEF B.

(Refer Slide Time: 05:37)

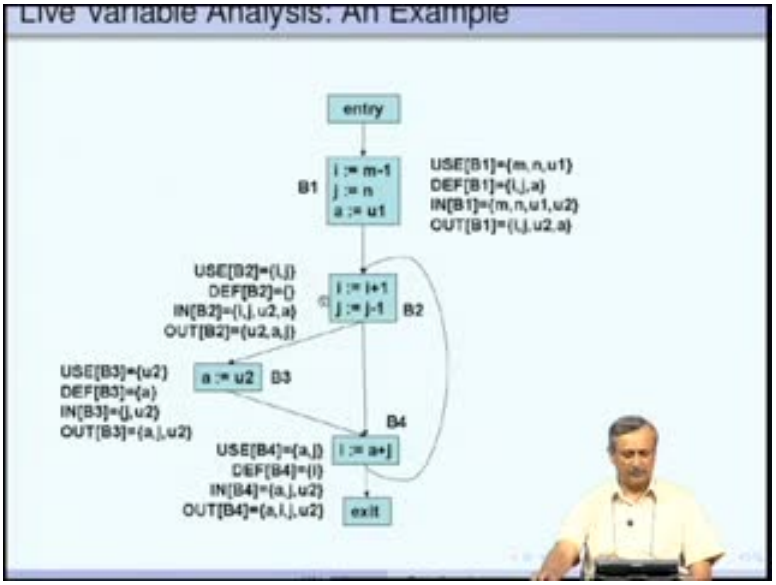


Let us understand why these are really correct. This is easy to understand. OUT B equal to union of IN S, see you are really looking at the backward flow analysis problem, so, when we compute the OUT of a basic block B, we are looking at the successors of B. For example, we want to compute the OUT of B2, we are going to look at the IN sets of B3 and B4. So, the idea is, if any variable is live here or here, it will be live here, because what we want is - just any path will do. There is a path along this direction and there is another path along this direction. So, we take the union of these two sets - that is the reason for the union. What about the IN set? So, again take the case of B2, it says, whatever is generated within the we are given OUT and we want to compute IN see we want we were given OUT and we are computing IN. That is why this is a backward flow analysis problem.

So, whatever is used before definition is definitely live at this point. In other words, because there is going to be a use, whatever values of i and j are defined here are being used in these two. That is why, i and j are live at this point, that is the used component and then there may be other variables which are live at this point, but they may or may not be killed. That is, if they are defined, then there is no way we can really use them.

So, if you look at the OUT of B2, it is u 2 a and j. So, DEF of B2 is 5. These three are not really killed. So, i j u2 and a are included in the IN set of B2. Let us verify it and see whether it is correct. i and j are of course correct, because i is used here, and j is used here, value of u2 is used here, there is no definition of u2 along this path and value of a is used here, and value of j, this j is different, j is defined here, but there is already a j here. So, we are only considering the variables and not the statement at which is used. So, u2 i j u2 and a are all live at this point. So, if you look at the IN set of B3, for example, you know u2 is definitely live at the entry point of B3 at this point, because it is visible straight away from the top, and let us take the OUT of B3 at this point, a and u2 a j and u2.

(Refer Slide Time: 05:37)

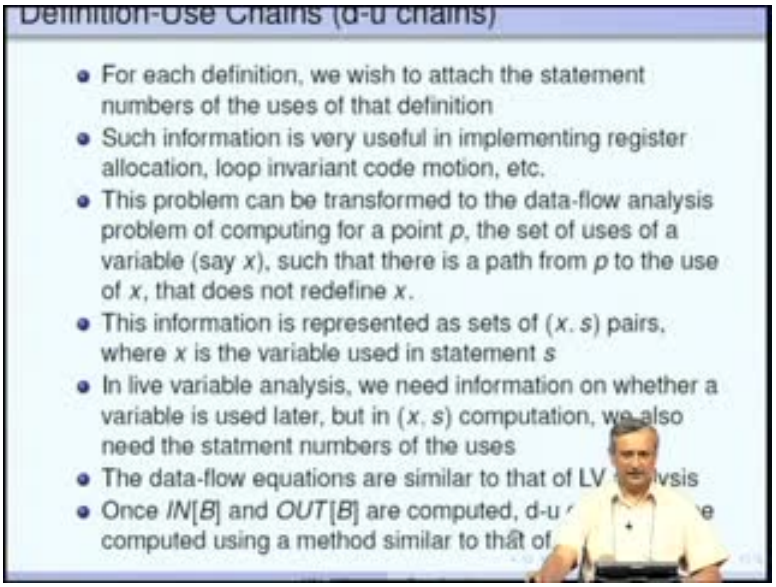


So, now you know the DEF contains, a, so, a is defined before any use. So, we are removing a, which is not live at this point, whereas, j and u2 are included in the IN set. That is how the IN sets for the live variables are computed. I hope this gives a flavor of how the backward flow analysis happens. It really does not matter I can do B1 first, then B2, then B3, then B4 and keep doing these computations of IN and OUT. USE and DEF are computed only once. We keep doing these computations of IN and OUT until all the values of IN and OUT stabilize.

So, the iterative analysis algorithm for backward flow analysis is very similar to what we had given for live variable analysis. I am not going to show that again. The same loop

with change etcetera is used. The equations are computed inside and then stabilization is achieved by changing the variable change.

(Refer Slide Time: 12:16)



So, that is as far as live variable analysis goes. There is another information that is very useful, these are the d-u chains or the definition use chains. What is the d-u chain? For each definition, we should consider all the uses of that particular definition. What do you mean by uses of a definition? Each definition actually defines a variable and then we are looking at the uses of the variable involved in that definition, but then you know that particular variable should not be redefined before usage, so that is what we really mean by uses of that particular definition.

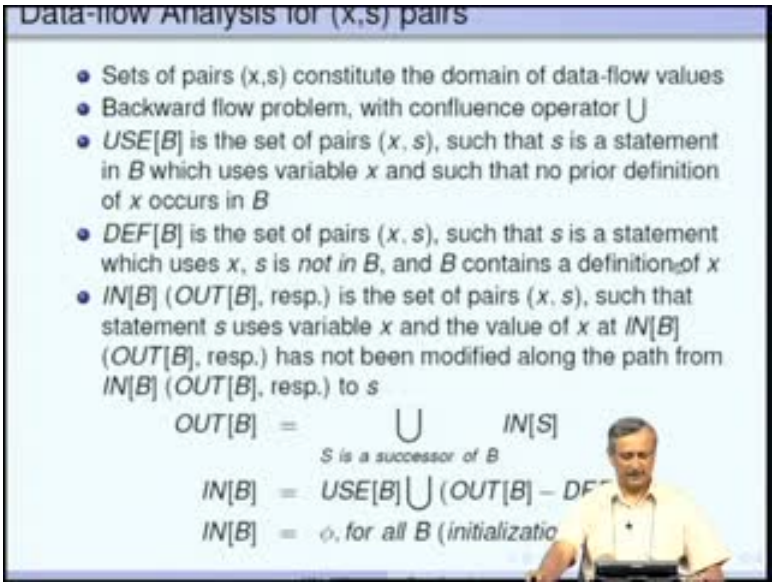
Such information is very useful in implementing register allocation loop, invariant code motion etcetera. We want you look to at the uses of a definition, you just have to go down the chain from the static from the definition and then you get the all the uses. So, this is a very useful piece of information, we have seen that the d-u chain information is very useful in creating webs and then doing register allocation via coloring.

This problem cannot be solved using live variable analysis, the reason is simple. Live variable analysis actually computes sets of variables only, it does not worry about the statement numbers where the variable is used, whereas, in the case of definition use chains we want to use the statement numbers also.

So, the problem of computing d-u chains can be transformed to a new dataflow problem which is very similar to that of live variable analysis. So, this problem is that of computing for a point p, the set of uses of a variable, say x such that there is a path from p to the use of x that does not redefine x. It is similar to the live variable definition. It is just that we are looking at the statement numbers also, that is why the sets of uses come in to picture.

So, we are associating with each x, a statement s also. The information is represented as sets of (x,s) pairs where x is the variable used in statement s. So, the dataflow equations are very similar to that of L V analysis. I am going to show you those. Once the IN and OUT sets are computed, d-u chains can be computed quite easily like the u-d chains.

(Refer Slide Time: 15:09)

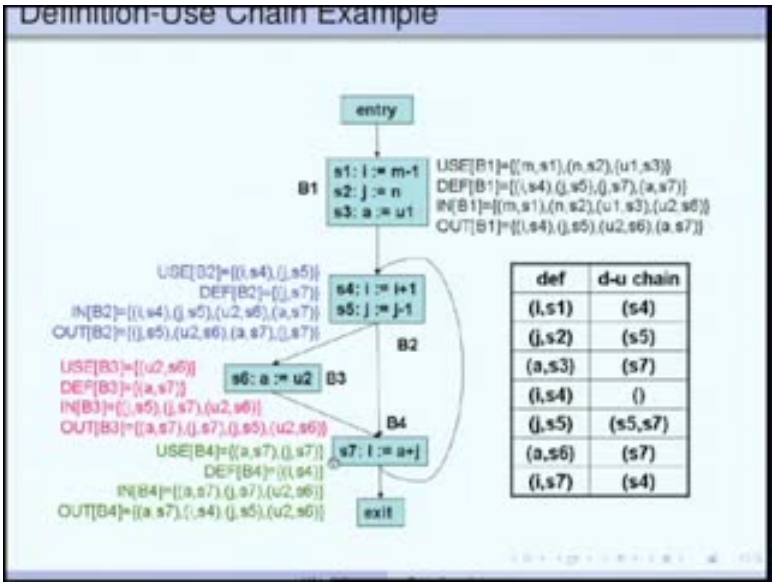


In this problem, the sets of pairs x comma s constitute the domain of dataflow values, whereas, in the live variable analysis, it was just the sets of variables. This is also a backward flow analysis problem and the confluence operator is union exactly like in the live variable analysis problem.

What about the gen and kill? That is the USE and DEF, so if there is a slight modification USE B is the set of pairs x s, such that there is a statement of in B which uses variable x and such that no prior definition of x occurs in B.

Here as I said, we are tagging along the statement number of the use also where we only worried about the use before the definition. DEF is similar. DEF B is the set of pairs x comma s such that s is a statement which uses x, but s is not in B and B contains a definition of x. So, B contains a definition and it obviously kills all other uses which are outside the basic block B. You may recall the reaching definitions. So, that is the way, we are doing it here. Look at all uses of x and the s should not be the statement in which it is used is not in B and B contains a definition of x.

(Refer Slide Time: 16:51)

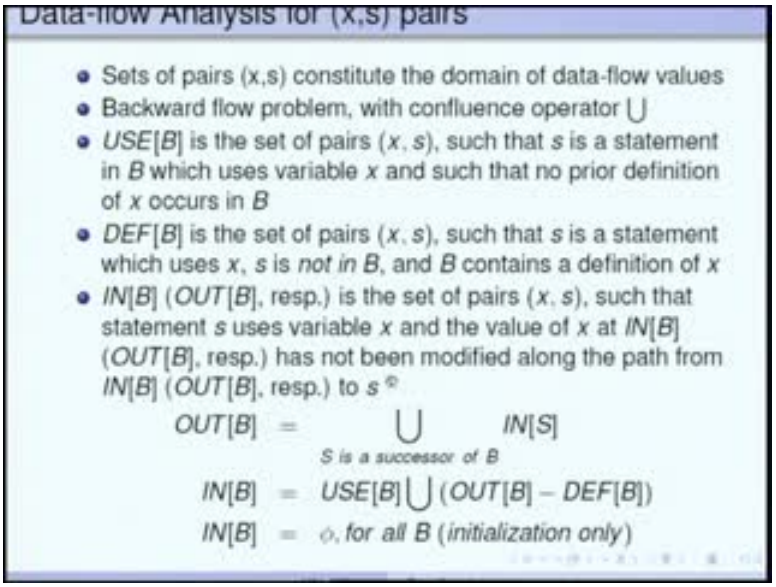


Let us see small example here, the same live variable analysis problem. We have numbered the statements as s1, s2, s3, s4, s5, s6, s7. We are looking at use before definition, so m comma s1, n comma s2, and u1 comma s3, very simple. Here it is i comma s4, and j comma s5, here it is u2 comma s6 and finally, here it is a comma s7 and j comma s7.

The DEF part, i is defined here, so it is deemed to kill all others in these blocks. i comma s4 - that is the statement which is outside the block which uses i j comma s5, another one, j comma s7, third usage and a comma s7 fourth usage. These are the four usages of the variables here. They are all defined here. They are all killed that is the DEF part. Whereas here, DEF of B2 is j comma s7, so j is being defined here, and j is the usage here, so that is why the DEF part comes here. Now, i part does not enter this DEF, because for this particular definition, there is no other usage that is going to be killed.

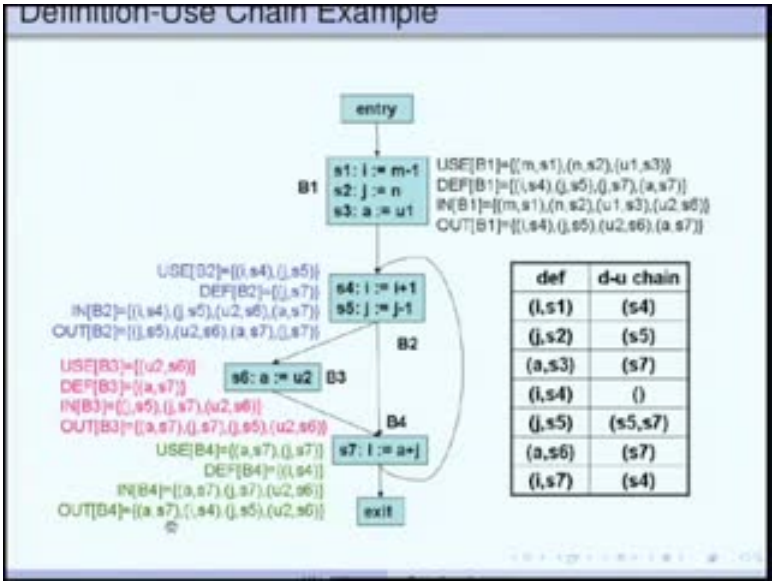
That is why this is not being considered. DEF of B3 - we are assigning a here – so, s7 uses a, so that is why a comma s7 is here.

(Refer Slide Time: 19:30)



So, that is the difference. When DEF of B4 is i comma s4, s4 is here, so i is defined here, so i comma s4 is this particular thing. So, this is the usage of I, this is the definition of i. If you look at this i, there is no other usage of i, you know these all are definitions this usage is not considered, because it is within. That is why i comma something was not included in DEF of B2. This is how we compute the DEF sets and USE sets. Now, let us look at the equations - OUT B is union of the IN sets of the successors of B and IN B is USE B union OUT B minus DEF B. These are identical to the live variable analysis problem. So IN B and OUT B are the sets of pairs x comma s such that the statement s uses the variable x and the value of x at IN B or OUT B has not been modified. It is very similar to the definition of this thing.

(Refer Slide Time: 20:03)



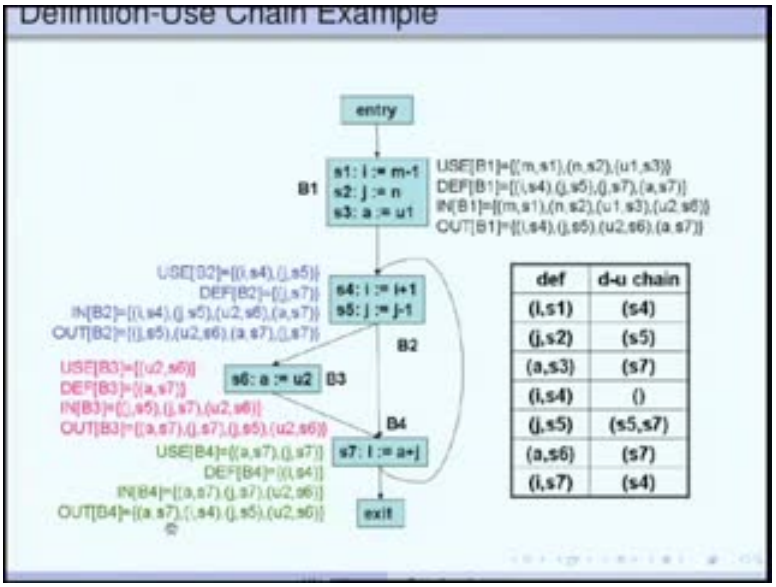
Let us look at the OUT of B1. i comma s4, j comma i comma s4 is here - so this is the usage. Let us say, the value of i that is live at this point is that of s1. So, i comma s4 uses this and then the value of j that is relevant here is that of s2, so that is being used here and then you know u2 comma s6 whatever was the initial value of u2 is used here, and then a comma s7, so, whatever is defined here is used here.

So, there is another path through which is redefined, but there is also one path through which it is defined. So, that is why, the OUT B contains B1 contains all these sets. IN of B1 is here, so, obliviously m comma s1, m u is used in s1, n comma s2, and u1 comma s3 are in this particular set IN and then a comma s7 is here, u2 comma s6 - whatever was the initial value of u2 is used here. So, these are the only sets which actually get in to IN of B1.

Similarly, we will take just one example, say OUT of B4, say OUT of B4 ,will contain a comma s7, i comma s4, j comma s5, and u2 comma s6 - why because of this arc - so a comma s7, so the a comes back through this path, i comma s 4, it goes there, j comma s5 that is again goes here, and u2 comma s6 which again travels here - so these are all in the OUT set. IN set contains a comma s7, j comma s7, and u2 comma s6. It does not contain i, because i is actually defined here, therefore, there is no way, this old value of i which is relevant here can go back.

This is how the IN and OUT sets are computed. I have to show you how the d-u chains are computed. This is a very simple job. Let us take each definition, i comma s1, j comma s2 etcetera, the information of OUT sets that is computed is used in computing the d-u chain. Take i comma s1 that is here, so, for this, what is the d- u chain? It should actually be s4, because that value is used in s4, but how do we actually get it? Look at the OUT set of B1, let us set this point. It is the only definition here is i comma s4 and then along this path. So, whatever is here, there are no other definitions of i here, so, we attach i comma s4 to this particular s1, so that is s4. How about j comma s2? It is very similar. j comma s5 is in the OUT set, so, that is the only one which is possibly a usage. There are no other definitions of j after this particular statement j equal to n. So, s5 is attached.

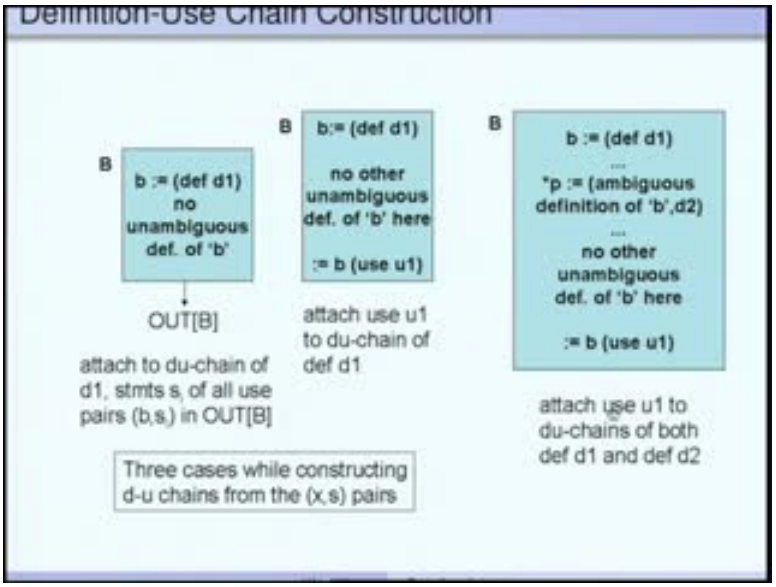
(Refer Slide Time: 20:03)



Similarly, s7 is attached to a comma s3. Let us look at i comma s4, i comma s4 has many of them, but it has no pair corresponding to i. That is because the variable value which is here is redefined at this point and this is not the same as that defined in s4. So, whatever is defined in s7 is used here and not what is defined in s4, that is why the d-u chain of i comma s4 is 5. What about j comma s5? This is different. See there is no other definition of j, so look at the OUT set of B2, it contains j comma s5, and j comma s7, so j comma s7 is here, and so, it is included in to the d-u chain from this particular thing. Because of this path, the value of j here is what is computed here, so j comma s5 also gets in. So, s5

and s7 are both on the d-u chain. For a comma s6, which is here - the OUT set contains a comma s7 that is the only one which is attached to this.

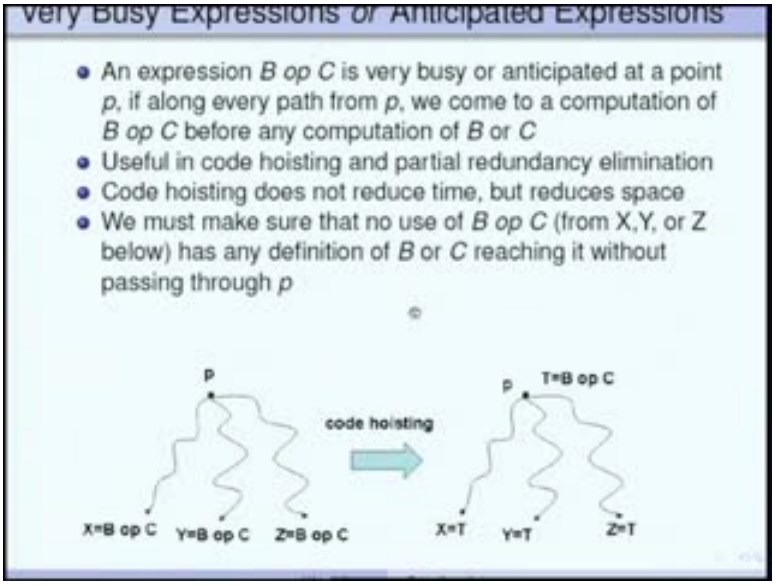
(Refer Slide Time: 25:07)



i comma s7 contains s4, that is here, so that is the one which is included - which is in the OUT set. So, this is formulized in this particular figure here. If there is a definition of B, that is DEF d1 and there are no unambiguous definitions of B here, till the end of the basic block, then attach to d-u chain of this d1- all the statements s of all the use pairs B comma s i in OUT of B - that is what we did here- we took all of them corresponding to the variable B and attached them to the definition d1.

So, if you take the second case, then there is a definition of B here, and then the usage of B here, and there are no other unambiguous definitions of B inside. So, this definition now gets only this use, so, attach use u1 to d-u chain of d1.This is the definition B and there is another unambiguous definition of B, let us say d2 and then there are no other unambiguous definition of B here, so, here is a use of B - because this is ambiguous, we have no idea whether it is correct, it is going to happen or it is not going to happen. So, we do not want to miss out on the information. We are going to attach the use u1 to both these d1 and d2. That is what I said here.

(Refer Slide Time: 26:37)



Very busy expressions or anticipated expressions. This is the next dataflow analysis problem that we are going to tackle. For example, consider this picture, there is a point p, then from along every path starting from p, we have x equal to B op C, y equal to B op C and z equal to B op C. B op C is being computed along every path starting from p.

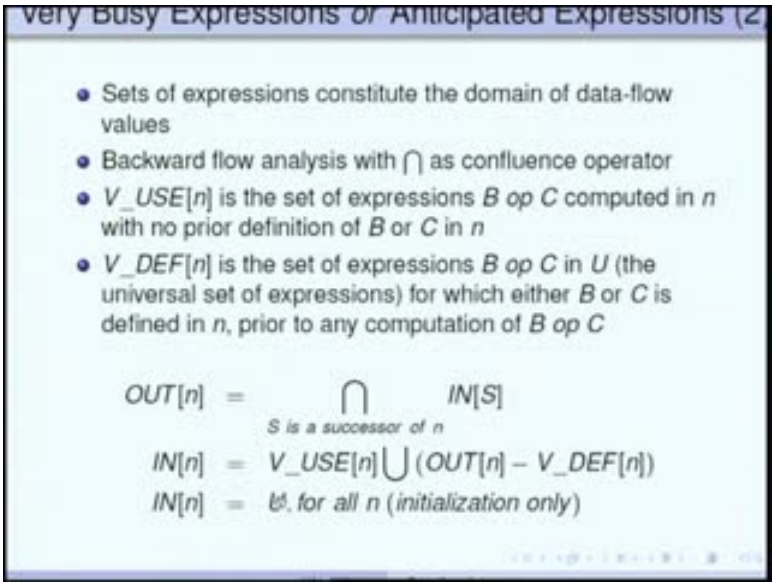
Such an expression B op C is set to be very busy, if there are no definitions of B or C along any of these paths. An expression B op C is very busy or anticipated at a point p, if along every path from p, we come to a computation of B op C - before any computation of B or C. This very busy expression is useful both in code hoisting transformations and also in partial redundancy elimination transformation.

What is code hoisting? It says, if you know B op C is very busy at p, why it should be really computed here you might as well compute it here, so this is the code hoisting that take place for B op C. So, we compute T equal to B op C right here in to a temporary T and then just assign x equal to T Y equal to T and Z equal to T.

There is no saving in time here in this code hoisting, because B op C is computed exactly once. But there is saving in space, the code for B op C need not appear in three places, but we must make sure that there is no redefinition of either B or C along any of these paths and secondly we must make sure that no definition of B or C reaches any of these things without passing through p.

In other words, if there is another definition of B which comes here and one comes through this path, then the B op C here will be very different from any of these. So, that should not happen.

(Refer Slide Time: 29:14)



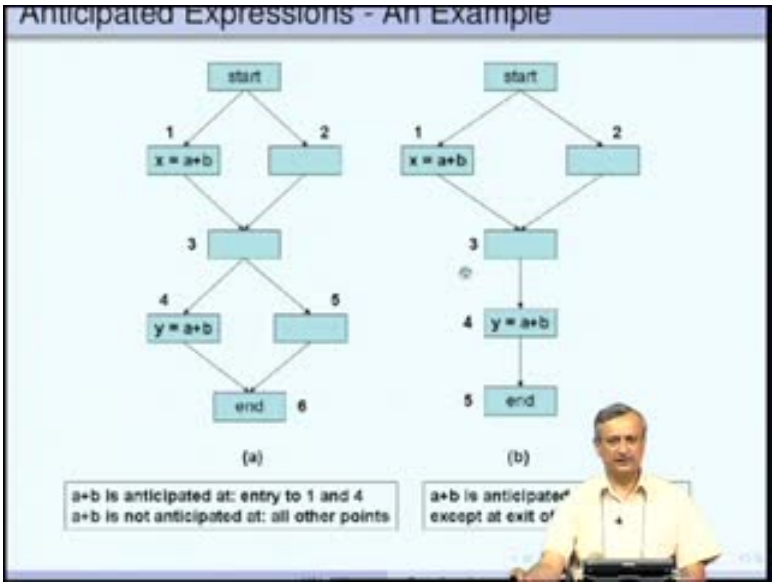
So, how do we define a dataflow analysis problem for very busy expressions? As usual, the sets of expressions constitute the domain of dataflow values here, we have seen that already and this is a backward analysis problem - why? Look at the equation, IN is defined in terms of OUT, the confluence operator is the intersection, so OUT n is intersection of IN S, where S is a successor of n. Why are we using n here - just that B op C is the expression we are looking at so using B here would be confusing that is only the reason.

What about V USE and V DEF. V USE corresponds to the gen set and V DEF corresponds to the kill set. So, V USE is the set of expressions B op C computed in n with no prior definition of B or C in n. This is very similar to the live variable analysis problem where we had a usage before any definition. This is a computation of B op C before any definition of B or C. What about V DEF? The set of expressions B op C in the universal set U of expressions, for which either B or C is defined in n prior to any computation of B op c. So, B or C, one of them is given a value and then B op c is computed. So, that constitutes the kill set because what happens is, once B or c is defined in the basic block, then the previous value of B op C is not very useful to us any more.

That is why when we look at this equation computing IN V USE, we include in the IN set, because these are visible expressions, B op C computed in n without prior definition of B or C.

Whereas, OUT n minus V DEF make sure that all those expressions whose variables were defined first and then the expression computed are kind of killed. They are removed from the OUT set. Then, as in any problem with intersection as a confluence operator, the IN is always initialized to the universal set for initialization purposes. This gives us a much bigger set for IN and OUT rather than compared to when we use five.

(Refer Slide Time: 32:09)

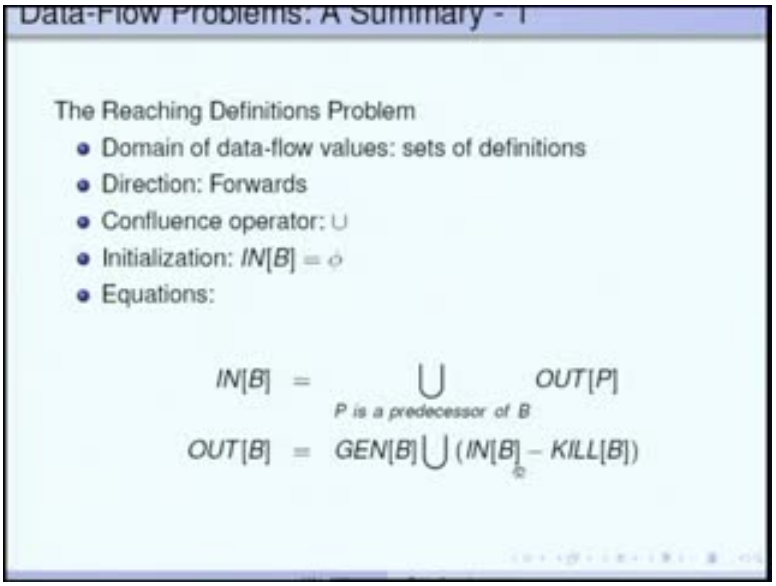


Here is a very simple example of anticipated expressions or very busy expressions, if you consider the expression a plus b that is the only one here, so at the point 1 and also the point 4 a plus b is anticipated, because a plus b is inside the basic block, it is being computed and we have not defined a or b here.

If there had been a definition of a or b just before this computation, then this would not have been anticipated at this point. That is how it is. At any other point, a plus b is not anticipated. Why? Let us take say 3, along this path a plus b is computed - no problem, but along this path a plus b is not computed; that is the reason it is not anticipated either at this point or this point or any of these points for that matter.

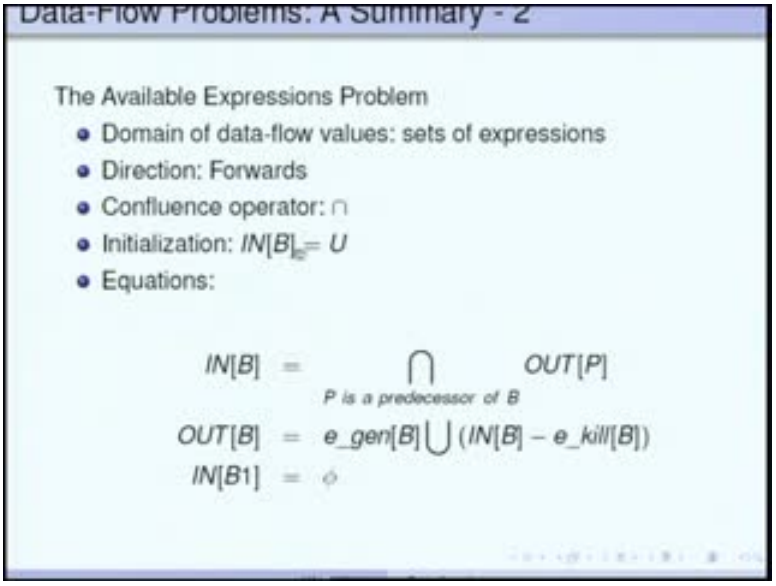
The situation is very different here. You take a plus b; look at any point there is computation of a plus b follow in along all paths. Here of course, a plus b is already computed inside and it is not killed, so it is anticipated at this point. If you take this point, there is only one path leading out, and along that path a plus b is computed again, so along this path there is anticipation of a plus b and at any point here this or this or this, we can take this, we can of course take only that path and a plus b is computed at this point.

(Refer Slide Time: 33:59)



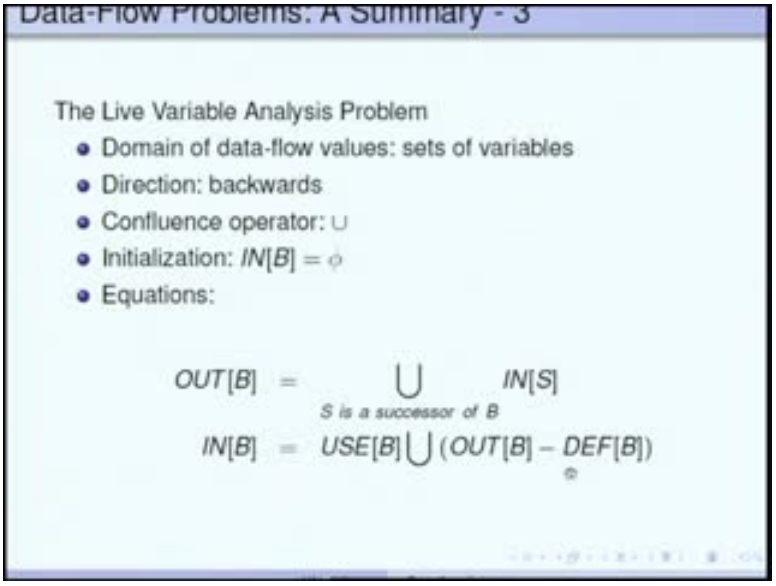
So, this makes it anticipable at every point in the flow graph, we will see how to use anticipated expressions in partial redundance elimination in one of the later lectures. Let us summarize dataflow analysis problems that we studied so far, we first studied the reaching definitions problem - the domain was sets of definitions, direction was forward, confluence operator was union, initialization was IN B equal to phi, and equations were IN B equal to union of OUT P where P is a predecessor of B, OUT B is GEN B union IN B minus KILL B.

(Refer Slide Time: 34:27)



Then we studied the available expressions problem - sets of expressions were the domain of dataflow values, forward problem, confluence operator intersection, initialization IN B equal to U, and equations were IN B equal to intersection of OUT P, P being a predecessor of B and OUT B is e gen B union IN B minus e kill B, IN B1 was always phi, and the initialization for all other blocks is U.

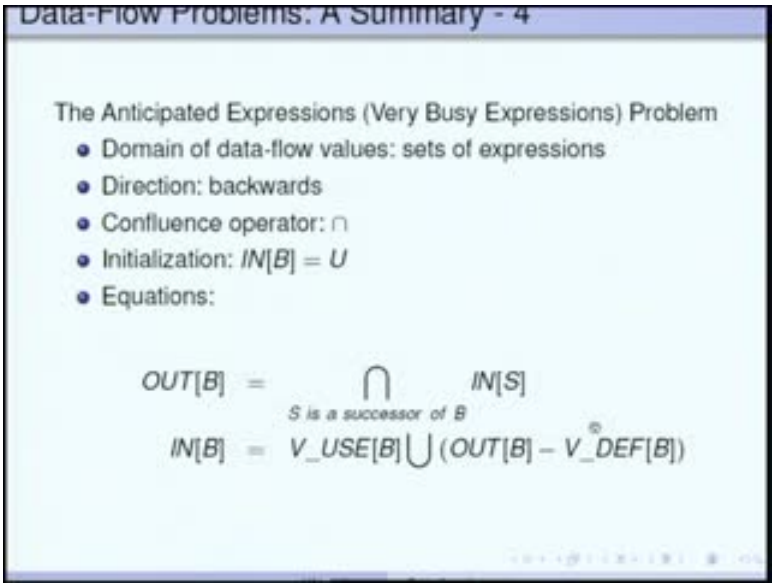
(Refer Slide Time: 34:59)



The live variable analysis problem one in which the dataflow values are all sets of variables and direction is backwards, confluence operator is union, the initialization is IN

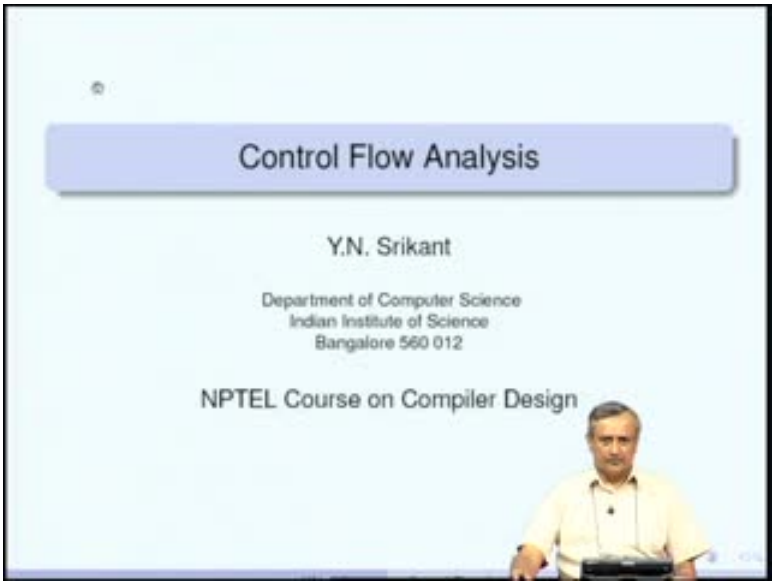
B equal to phi, equations are OUT B equal to union of IN S, S being a successor, IN B equal to USE B union OUT B minus DEF B.

(Refer Slide Time: 35:26)

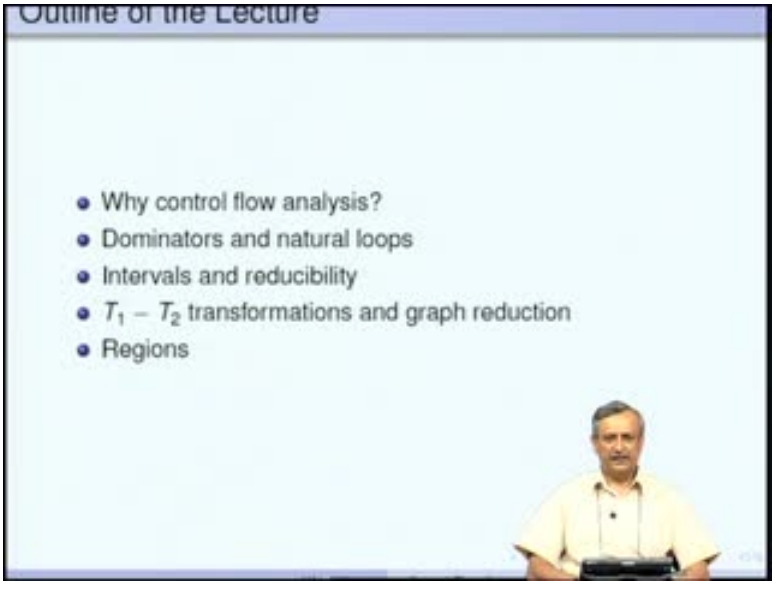


The last one in the matrix, the fourth combination anticipated expressions we just now studied: sets of expressions constitute domain of dataflow values, backward problem, intersection confluence operator, IN B equal to U being initialization, OUT B is intersection of IN S, S being a successor and IN B USE V USE union OUT B minus V DEF B. So, this constitutes a summary of the dataflow problems. Now, will move on to control flow analysis.

(Refer Slide Time: 35:58)

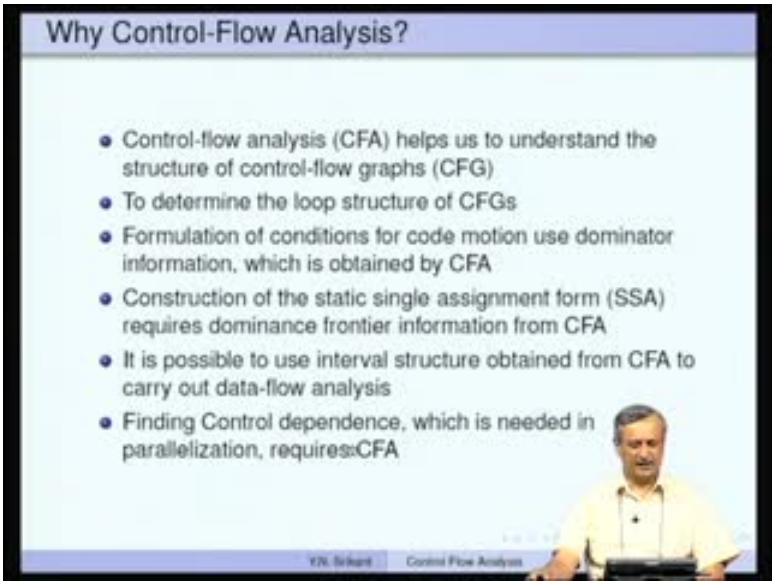


(Refer Slide Time: 36:05)



Welcome to the lecture on control flow analysis. In this lecture, we will see why control flow analysis is required and then we are going to study dominators and natural loops, intervals and reducibility, T1 T2 transformations and graph reduction, and then the concept of what are known as regions.

(Refer Slide Time: 36:30)



The dataflow analysis that we studied in the last lecture tells us how the values flow through the control flow graph. The control flow graph first was given to us, but then we only talked about loops, branches and things like that in the control flow graph, but there was no structure of the control flow graph such as loops and branches that we understood formally.

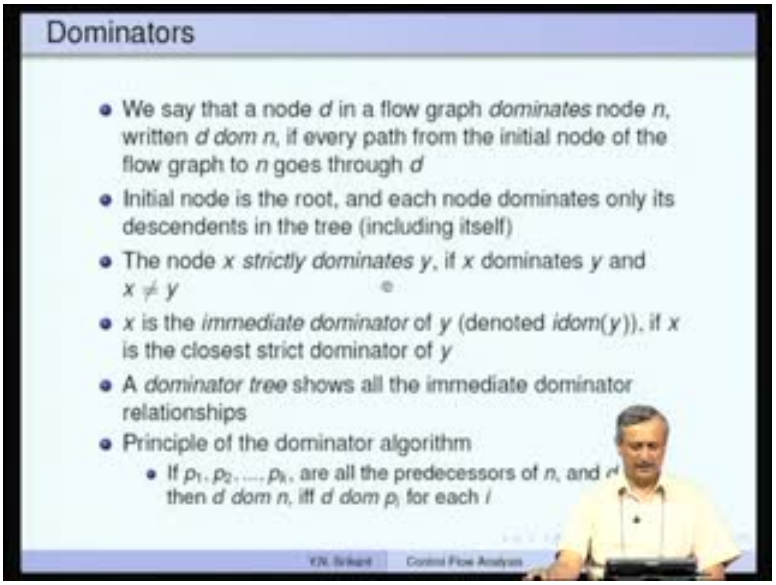
Control flow analysis really helps us to understand the structure of the control flow graphs. For example, we want to formally define what exactly are the loops in a control flow graph and if we do not do this, we will not be able to do the optimizations which are meant for loops. So, formulation of the conditions for code motion - these use the concept of what are known as dominators. So, dominator information is computed by control flow analysis.

Then, I mentioned about the intermediate form called static single assignment form. The static single assignment form requires information known as dominance frontier. Control flow analysis is going to compute this information. We will study dominance frontiers along with the static single assignment forms

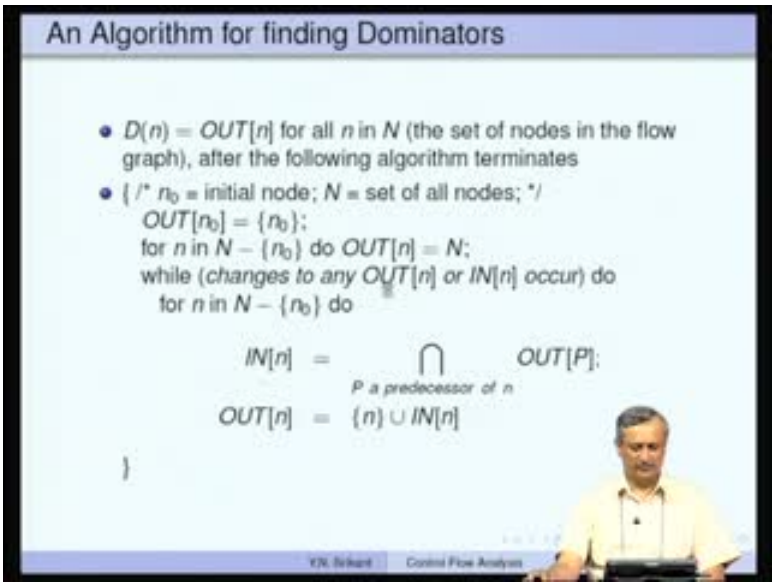
Then, control flow analysis also tells us how to determine the interval structure of a control flow graph and use this interval structure for carrying out dataflow analysis. The dataflow analysis we did so far is called iterative dataflow analysis, whereas, there is another school of sort which uses interval based dataflow analysis. Intervals structure

enables such interval based dataflow analysis. Control dependences, which are essential for parallelization and concurrentisation, they require control flow analysis dominance frontier, dominator information, etcetera. So, these are the reasons why we must understand control flow analysis thoroughly.

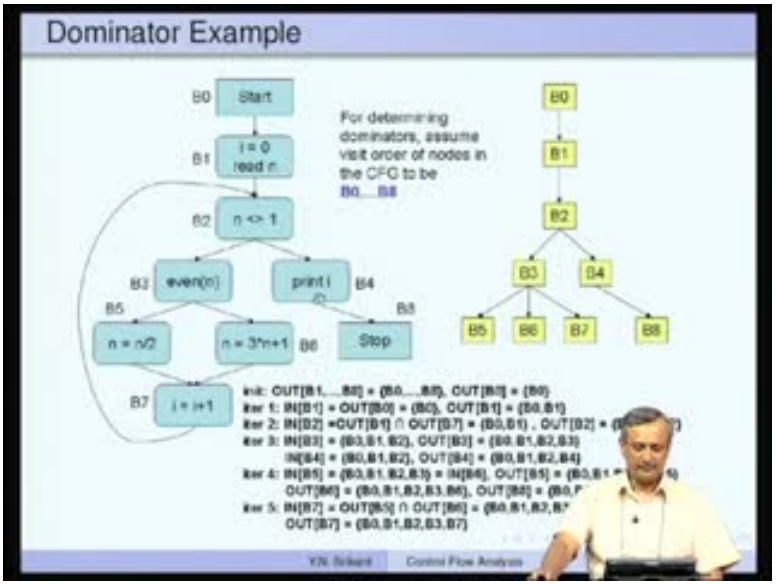
(Refer Slide Time: 39:17)



(Refer Slide Time: 39:47)



(Refer Slide Time: 39:48)

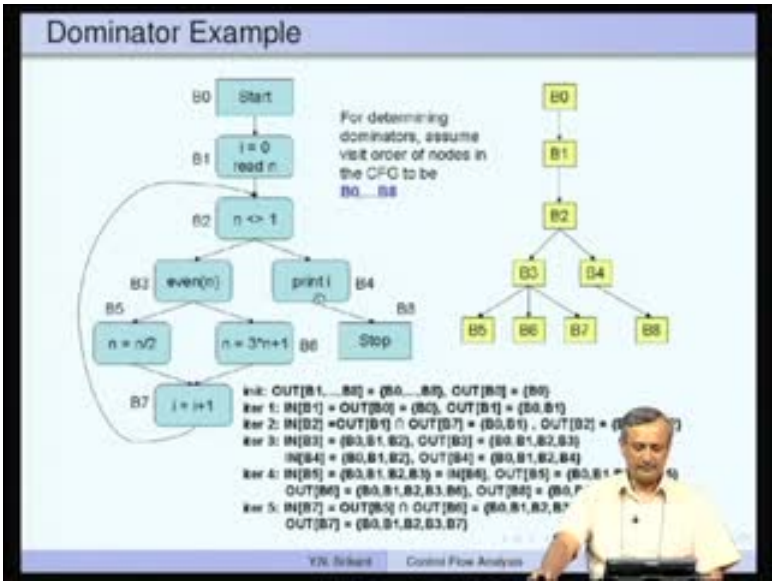


Let us begin with the most important concept of dominators. What exactly is a dominator? A node d in a flow graph dominates a node n and we write it as d dom n, if every path from the initial node of the flow graph to n goes through d, let me show an example and explain. The question to be asked is, does B1 dominate B6? The answer is yes. The reason is, we are going to look at the paths from the initial node that is B naught and then to the node B 6, and make sure that every path through to from B naught to B6 passes through B1, so from here we go like that and reach B6, so we must compulsorily pass through B1.In fact B2 also dominates B6, B1 B naught also dominates B6. But B4 definitely does not dominate B6, because there is no path to B6 itself. Actually B7 is very interesting. Neither of B5 and B6 dominate B7, why? Let us take the definition again, it says, starting from the initial node going to B7, every path must pass through B5, now, if I say B5 dominates B6, I will take the path from B naught pass through B1, B2, B3, B6, and reach B7, so I by-passed by B5. But if I say B6 dominates B7, I will take the path starting from B naught pass through B1, B2, B3, B5 and then B7, I have by-passed B6. But B3 definitely dominates B7, because no matter what, I always have to pass through B3 and then go to B5 or B6 and reach B7, so B3 definitely dominates B7.

The initial node is the root and each node dominates only its descendants in the tree. So, we are going to represent the dominator information in the form of a dominator tree. This is the root of the tree, this is the initial node which always dominates all the nodes in the tree and the way the tree information is maintained, each node dominates only its

descendants. For example, B1 dominates all these nodes, but it does not dominate B naught, you can easily see that B3 dominates B5, B6, and B7, so all these three are children of B3 and therefore, you know this dominator information is very compactly represented in the tree.

(Refer Slide Time: 39:48)

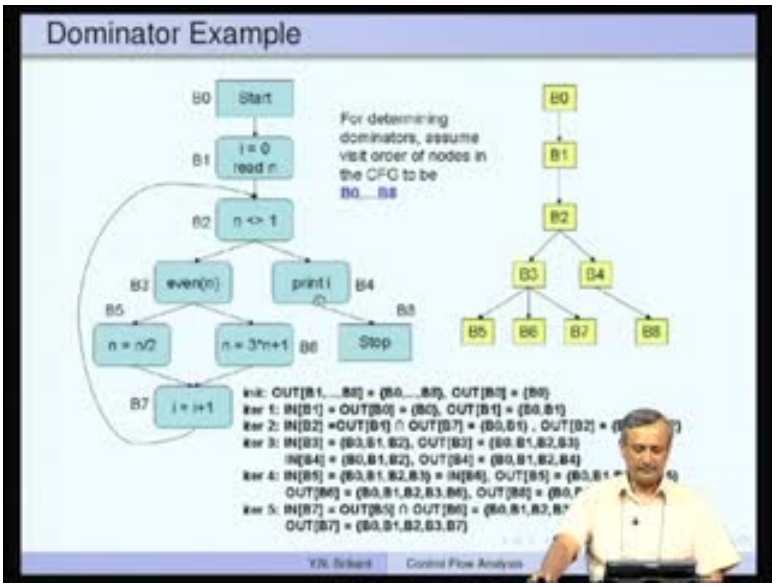


The node x strictly dominates y, if x dominates y, and x is not equal to y - self dominance is eliminated then it becomes a strict domination. x is the immediate dominator of y denoted i dom y, if x is the closest strict dominator of y.

In other words, if you look at this - B2 immediately dominates B3, B2 immediately dominates B4, but B2 does not immediately dominate B6, whereas, B3 immediately dominates B5, B6, and B7, so the tree actually shows only the immediate dominator relationship. That is what is said here. What is the principle of the dominator algorithm? If p1 to pk are all predecessors of i node n, and d is not equal to n, then d dom n, if and only if, note that this is a necessary and sufficient condition d dom p i for each i. So, if these are all the predecessors, then the node d must dominate all the predecessors otherwise d will not dominate n. Let us check that out. This B7 has two predecessors, all the others we will see later. B2 also has two predecessors, so there are two nodes B5 and B6, which are predecessors of B7. So, now if some node has to dominate B7, the result says, the same node must dominate both B5 and B6. This is true. Look at the tree here-

so B3, B2, B1 and B naught all of them dominate B7, for example, if B3, B5 and B6 do not dominate B7. That is why this information is not true.

(Refer Slide Time: 39:48)



So, if one of them actually were dominating this, then that would have been dominated by these. Let us take B2, its predecessors are B1 and B7, let us take B2 here, only B1 dominates B2, because B1 dominates both the predecessors of B2, that is B1 itself and also B7.

This is how we use this dominator information rather the theorem in order to construct dominators. Let us look at the algorithm for constructing dominators. It is actually an algorithmic form of the same theorem. The notation is d n equal to OUT n for all n in N.

So, we compute OUT n and that is the dominator information of n. n naught is the initial node and N is the set of all nodes. So, initialization and permanent assignment OUT n naught is N naught and for n in N minus n naught OUT n equal to N. This is initialization.

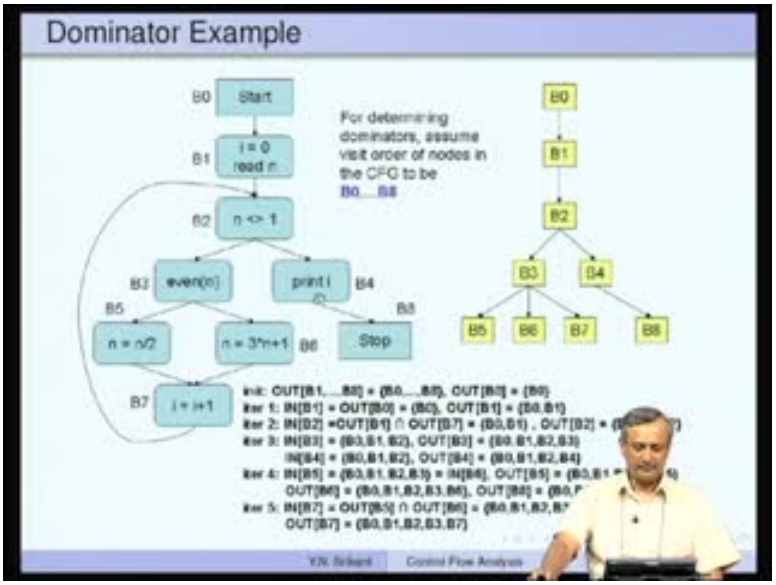
Then the set of dominators of n is what is given by d n. So, the set of dominators of n naught is just n naught. n naught is initial node, it can never be dominated by any other node.

So, while changes to any OUT n or IN n occur. This is very similar to dataflow analysis recall that. For all the nodes in N minus n naught do so IN n is this is the from theorem

intersection of OUT P, P a predecessor of p n and then OUT n is n union IN n. It is a very simple dataflow analysis.

Let us apply it on this particular example – initialization - OUT of B1 to B8 is B naught to B8 that is capital n all the nodes in the graph, OUT of B naught is B naught so just for this particular node and then iteration 1 IN of B1 - so this is IN these has only one predecessor. So, it becomes OUT of B naught that is B naught itself. So, then you know OUT of B1 is B naught comma B1, there is no other possibility, then initialization is very important here, because OUT of B naught has been initialized B naught, that is why IN of B 1 gets OUT of B naught and that is only one element

(Refer Slide Time: 39:48)

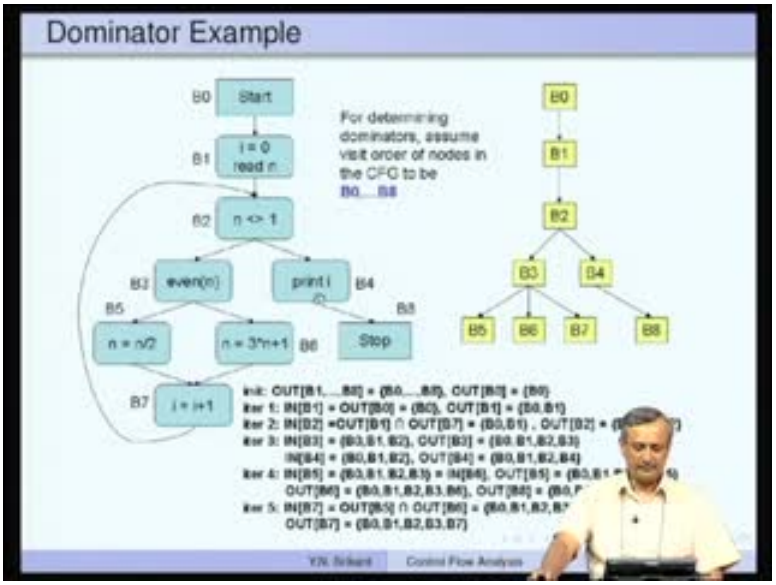


If this side also be initialized to capital N we would not have reached the solution at all. IN of B2 that is here it has two predecessors- one is B7, other is B1. So, we take the intersection, so OUT Of B1 is B naught B1, OUT of B7 is B naught to B8, of course the intersection gives us B naught B1 itself and the OUT of B2 is including this node, B naught, B1, and B2.

Recall that OUT n is n union in n. Similarly, we go on with iteration two, IN of B2 is over, IN of B3 - iteration three, IN of B3, there is only one node coming here, so it is very simple, it is OUT of B2 that is B naught, B1, B2. OUT of B3 is including this into the B naught, B1, B2, and B3.

Now, IN of B4 on this side is again the same B naught, B1, B2, and OUT of B4 include B4 into the set, so B naught, B1, B2, B4. Iteration 4 - IN of B5, again it has only one element, one edge coming from B3, so we get OUT of B3 and that is same as IN of B6 along this direction also and OUT of B5, you include B5 in to it OUT of B6 we include B6 in to it and on this side, OUT of B8 is just B naught, B1, B2, B4, and B8 along this direction.

(Refer Slide Time: 39:48)



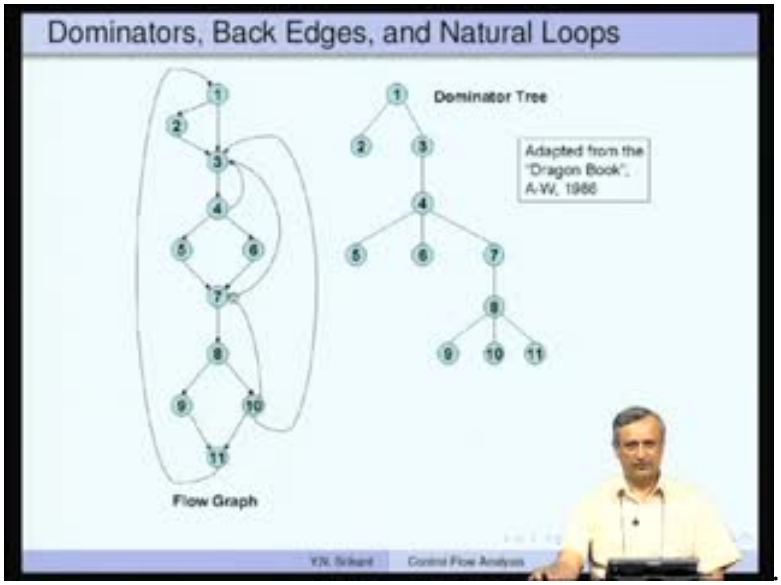
This is interesting. IN Of B7 must take the intersection of these two, so these two automatically go out. OUT of B5 IN 5 intersection OUT of B6, one of them contains B5, the other one contains B6, so OUT of B5 contains B5, OUT of B6 contains B6, but both of them do not contain B5, B6. So, it goes out.

The only once which remain or B naught, B1, B2, B3 and finally, OUT of B7 is B naught, B1, B2, B3 and itself B7. Now, for the second iteration, OUT of B7 has changed, so IN of B2 becomes OUT of B1 intersection OUT of B7, it is the same as B naught, B1. There is no change.

It so happens that the order in which we have visited everything has stabilized as we went along, but only thing is we made five iterations. If we had carried out from bottom or if we had not used this, we would have actually done even more number of iterations possibly.

So, we did this, then this, then this, then this, then this, and so on. What I am assuming is, we did this and all others did not change, then we did this and all others, none of them really changed and so on and so forth.

(Refer Slide Time: 50:52)

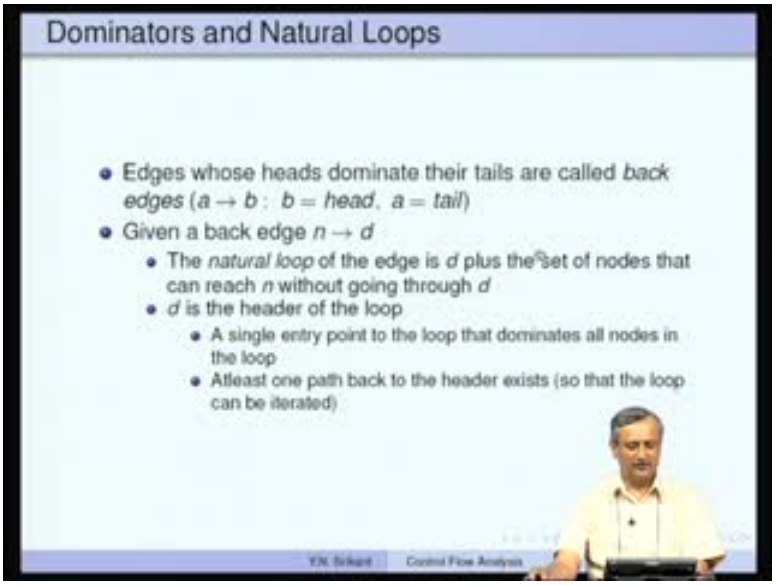


Now that we have computed the dominator information, let me show you another bigger example. This has been adopted from book which is called as a dragon book. A huge graph here and this is a start node. If you look at it, one dominates all nodes that is by definition. Why does 1 dominate 6? For example, 1 dominates 6 it says, obviously since 1 is the root, every path must start from 1. Let us look at something more interesting. 4 dominates 10 - that means starting from 1 all paths to 10 must go through 4, which is very obvious. We go like this and then branch this way or this way we go like this and like this.

That is very simple. The dominator tree is mentioned here - 4 dominates 9 and so on and so forth. We just look at the path starting from the start node and then we go on to that particular node if it is traversed in every path, then it is dominated by that particular node.

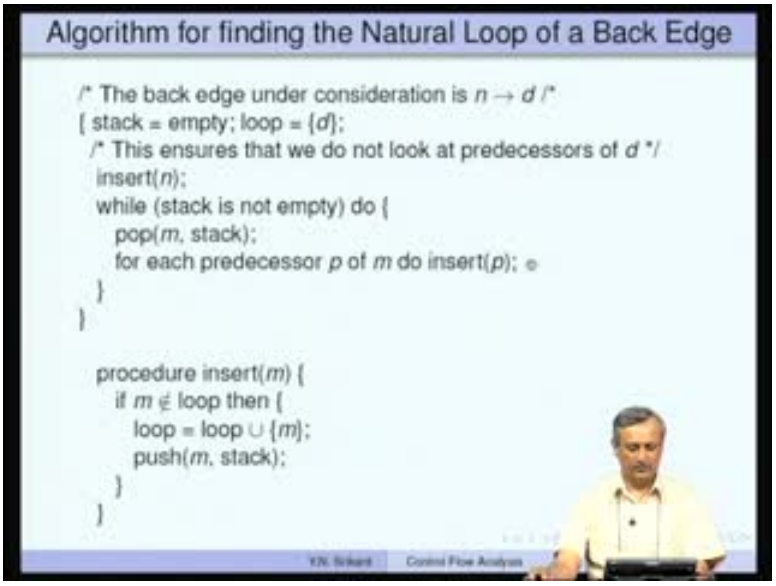
Here, we have changed the structure of the flow graph. Here we add this edge here, we had changed it to the other one, and the rest of edges did not change, so the dominator relationship also has not really been affected.

(Refer Slide Time: 52:40)

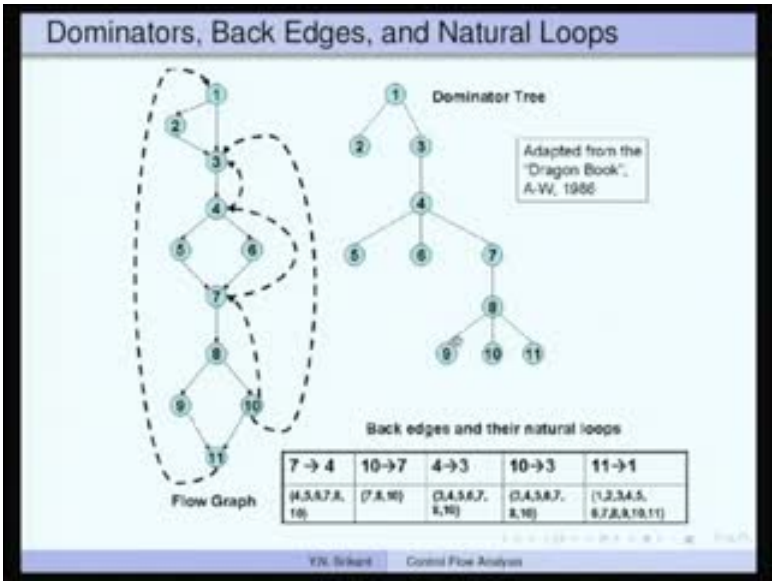


So, minor changes in the graph do not affect the dominator information. Now that we have defined dominators, we are ready to define what exactly are called back edges and natural loops. Edges whose heads dominate their tails are called back edges.

(Refer Slide Time: 53:00)



(Refer Slide Time: 53:01)



Let me show you a simple example. For example, in this tree, we know that 4 dominates 7, so there is an edge from 7 to 4, so that is called as a back edge. The back edges are all listed here, 7 to 4 is a back edge, that is because 4 dominates 7, 10 to 7 is a back edge, because 7 dominates 10, 4 to 3 is a back edge, because 3 dominates 4, 10 to 3 is back edge ,10 to 3 is back edge because 3 dominates 10, 11 to 1 is a back edge, because 1 dominates 11 all that is clear from the diagram. Given such a back edge, n to d, now we are ready to define the natural loop of a back edge.

Remember we do not know anything about the loop structure of the control flow graph we are now trying to discover such a loop structure, Loop structure is defined with respect to back edges. For every back edge there is set to be a natural loop.

The natural loop of the edge that is n to d is the node d plus the set of nodes that can reach n without going through d. In other words, n to d is the back edge, it goes backwards, so starting from d which is supposed to be the header of the loop we want to travel to n, so you look at all the nodes through which we can travel to n, but those nodes cannot be included again. That is what this says.

If you look at this, let us say we are looking at this particular 10 to 3, we start from 3, we want to go to 10 we look at all the nodes through which we can reach 10 and include those nodes, so 3, 4, 5, 6, 7, 8, 10, so these all are the nodes which we travel but what is not intuitive is the natural loop of the edge 4 to 3. That must be studied only with respect

to this particular algorithm. d is the header of the loop, the properties of such a loop are - there is a single entry point to the loop that dominates all nodes in the loop and at least one path back to the header exists that is along the n to d, so that is d is the dominator and n to d is the path back to the header.

At this point, we will stop this lecture and in the next part of the lecture, we will study an algorithm to find the natural loop of a back edge and then go on to the other uses of dominator information. Thank you!