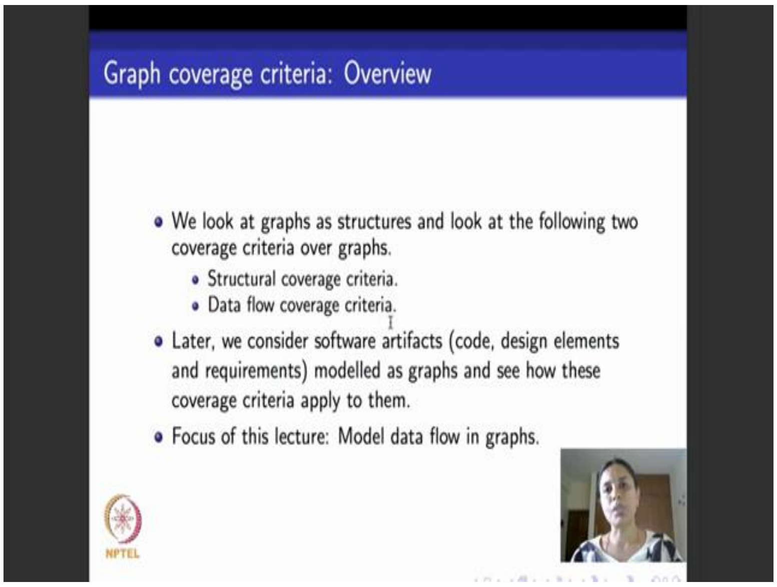
Software Testing
Prof. Meenakshi D’Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 11
Data flow in graphs

Hello again, from today’s lecture onward well continue with graphs, but we will begin to look at not only control flow and also about discuss about modeling data and how to write a test cases and define test paths that deal with handling about how data flows.

(Refer Slide Time: 00:30)



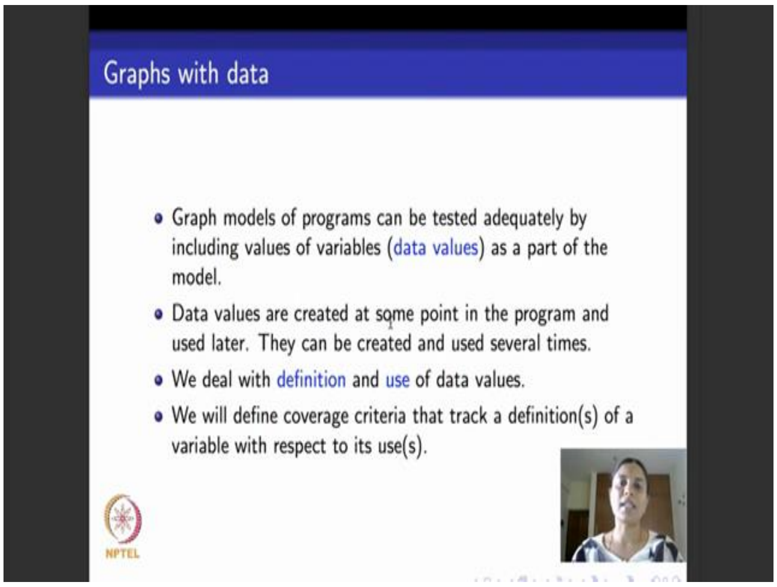
Graph coverage criteria: Overview

- We look at graphs as structures and look at the following two coverage criteria over graphs.
  - Structural coverage criteria.
  - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Model data flow in graphs.

Just to recap where we are in the course. We are looking at a test case generation algorithms and test requirement definitions, based on graph coverage criteria. In the previous week, we saw structural graph coverage criteria which is just coverage criteria based on the structure of the graph, vertices, edges, paths and so on. This week I will begin with dealing with a second part which is data flow coverage criteria and then after we do these two, very soon we will look at various software artifacts core design requirements, see how to model them as graph and recap all these structural coverage criteria and data coverage criteria that we saw and see how to use them to actually test these software artifacts.

So, the focus of today's lecture and for a couple of lectures more to go would be related to data flow coverage criteria. What we will be seeing today is what is data in a software artifact mainly programs as far as this is concerned and how to model data in graphs. In the next module I will tell you about once we have models augmented with data, how to define coverage criteria and what is the state of the art when it comes to defining test paths that achieve these coverage criteria.

(Refer Slide Time: 01:51)



Graphs with data

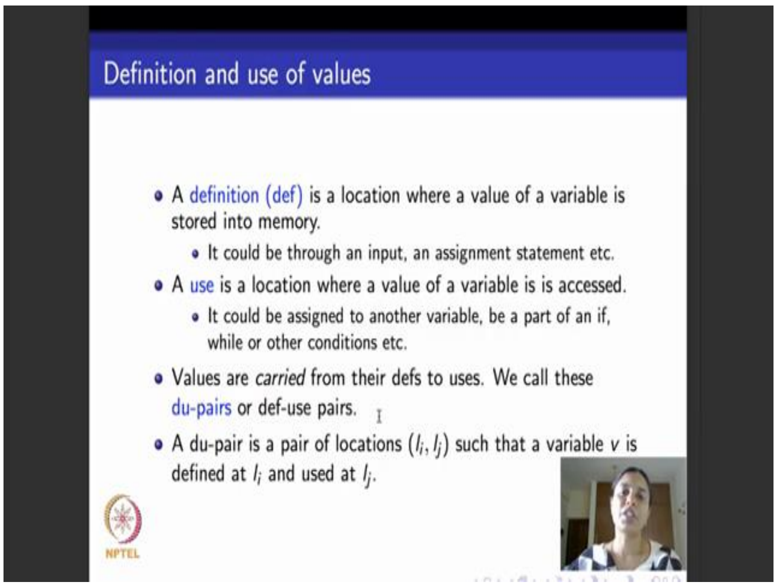
- Graph models of programs can be tested adequately by including values of variables (data values) as a part of the model.
- Data values are created at some point in the program and used later. They can be created and used several times.
- We deal with definition and use of data values.
- We will define coverage criteria that track a definition(s) of a variable with respect to its use(s).

So, what is data? As far as programs are concerned, as far as many software artifacts are concerned there is only one kind of data available. Data is available as variables. Variables could have types: variables could have simple types variables could be complex types like arrays several other user defined types and so on, but whatever it is all the data that typical program deals with is basically represented as variables. What are these variables used for in the program? They are used because they represents some kind of information or data, but there are basically 2 kinds of basic ways in which we subject variables to the program. One is to say that a available gets created at some point in the program typically when it is declared, when it is declared and initialized. And at some point later in the program a variable gets used. So, it could be the case that certain declarations get missed or certain use is get missed.

Suppose the variable that is not declared suddenly gets used in a program then; obviously, the program will not compile. It will be a compiler error, compiler will say that there is a variable that is not well declared in the program, but think of the other way round right a variable is declared in the program, but never used at all. Maybe the program was large and this programmer meant to use this variable. So, he declared it, but did not really use it because it was wrongly declared. We really do not want such things to happen.

So, we want to be able to track a variable from the place where it is defined to all the places where it is used. And we would not check whether it is used at all or not. So, what we will deal with throughout this lecture would be how to define data places where variables are used, and what are the places where they are actually put to use. And in the next lecture we look at coverage criteria that will track the definition of a variable with reference to it is use.

(Refer Slide Time: 03:53)



Definition and use of values

- A definition (def) is a location where a value of a variable is stored into memory.
  - It could be through an input, an assignment statement etc.
- A use is a location where a value of a variable is is accessed.
  - It could be assigned to another variable, be a part of an if, while or other conditions etc.
- Values are *carried* from their defs to uses. We call these du-pairs or def-use pairs.
- A du-pair is a pair of locations (l_i, l_j) such that a variable v is defined at l_i and used at l_j.

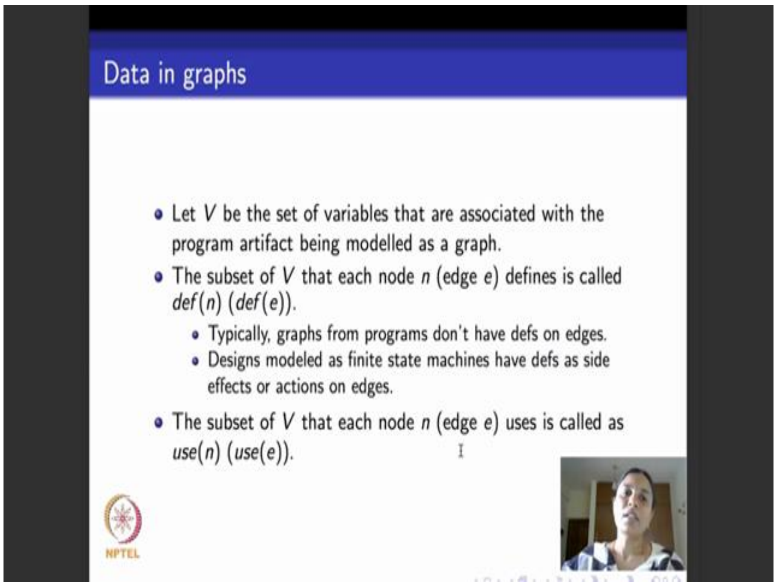
So, what is the definition of a variable? A definition of a variable is a location in the program where the value of the variable is stored into memory. That location could be an input statement right, it could be an assignment statement, it could be parameter passing from one procedure to another, could be any kind of statement. We will call statement abstractly as locations so that we can smoothly switch between using statement is a location and when we look at graphs nodes for as will become location. So, just to recap what is definition of a variable with use the word def for short; it is any location in the program where a variable is stored into memory it is value is stored into memory, assignment statements, input statements, parameter passing, procedure calls through parameter passing all these could be places where a variables value is defined.

Now, a defined variable has to be put to use. So, what is a use? A use is a location in a program where a variables value that is store is accessed is accessed by the statement of a program, by an assignment statement where and it could be assigned to another variable, it is accessed because it is being checked as a part of a condition that comes along with an if statement or it is checked, it is used as a part of a condition that comes along with the loop like for loop or while loop. So, whenever a value of a variable is accessed you say that that is a use corresponding to a variable. So, variable is defined and then in one or more ways and then a variable is used in one or more ways and typically a program carries the value of each variable from it is definition to it is use right. So, these pairs of definitions and uses of a variable are what are known as du-pairs or def use pair.

So, what is a du-pair a du-pair is a pair of locations say (l i, l j) such that a particular variable v is defined that l i and used at l j. Please note that even though ive not mentioned here v is an explicit parameter toward du-pair per variable they could be several du-pairs right, per pair of location there could be several variables that are defined and used at any point of time given a variable you are looking at what it is d u pair and given a pair of location, you are looking at all the set of variables that are involved in that location definition. And use of that location we may or may not mention it explicitly in this lecture, but there is always a parameter or an attribute to every definition or a use or a d u pair which is always a variable, because if variables are not there what do we define and what do we use.

So, what did we learn till now? We learned what a definition of a variable is. It is basically a place where if variable values first time written into memory: could be an input statement could be a declaration could be an initialization could be parameter passing several things what is a use a use of a variable is a a statement or a location where a value of the variable is accessed it is use could be because it comes on the right hand side of an assignment statement or it is used is a part of a predicate that comes for checking for an if or for a loop condition. What is a d u pair a du-pair is a pair of locations (l i, l j) such as the variable is defined an l i and used at l j?

(Refer Slide Time: 07:15)



Data in graphs

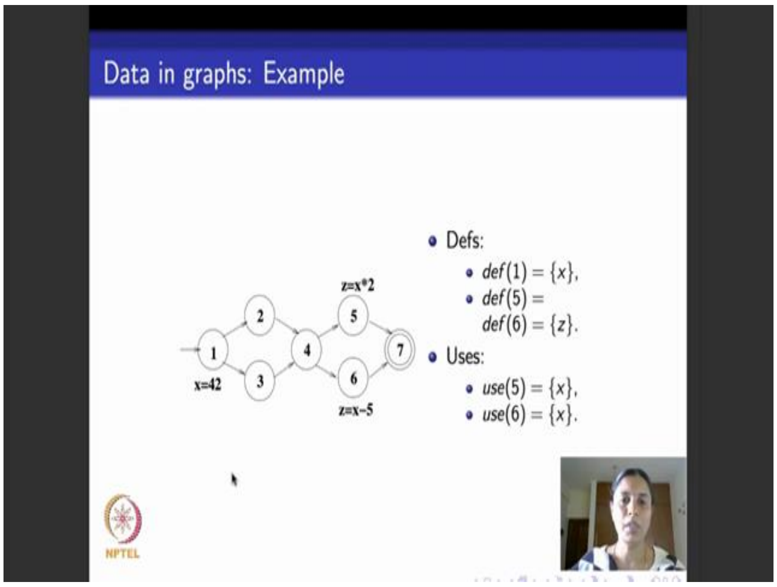
- Let V be the set of variables that are associated with the program artifact being modelled as a graph.
- The subset of V that each node n (edge e) defines is called $def(n)$ ($def(e)$).
  - Typically, graphs from programs don't have defs on edges.
  - Designs modeled as finite state machines have defs as side effects or actions on edges.
- The subset of V that each node n (edge e) uses is called as $use(n)$ ($use(e)$).

So, consider program that are let us say modeled as a graph. We will look at graph or data flow testing with reference to graphs in this module. So, we consider a program that is modeled as a graph, and let v be the set of variables that are associated in the program right. I want to know how these variables now come inside this graph right. So, what I do is I take the graph then graph has nodes and edges and per node and per edge, wherever it is applicable, I will say which are the variables that are defined in that node called as def of n for a node n. And for an edge e I will say which are the variables that are defined in the edge e.

Typically graphs that correspond to programs will not have definitions on edges. I mean will not have definitions on edges, as I told you. When is a definition, the definition is when a variable is written into memory. So, it is like having an assignment statement is a part of an edge, it is rare, it is rare or almost impossible in a graph that models a program. But for graphs that come as design models let us say graphs that come as finite state machines modeling designs, it is possible to have definitions on edges. Definitions could be thought of as actions that change the value of a variable or side effects related to an edge right. The subset of the set of variables that each node or edge uses is called use of n or use of e.

So, per node and per edge in the graph, we talk about what are the subsets of variables that it defines and what are the subsets of variables that it uses.

(Refer Slide Time: 08:47)

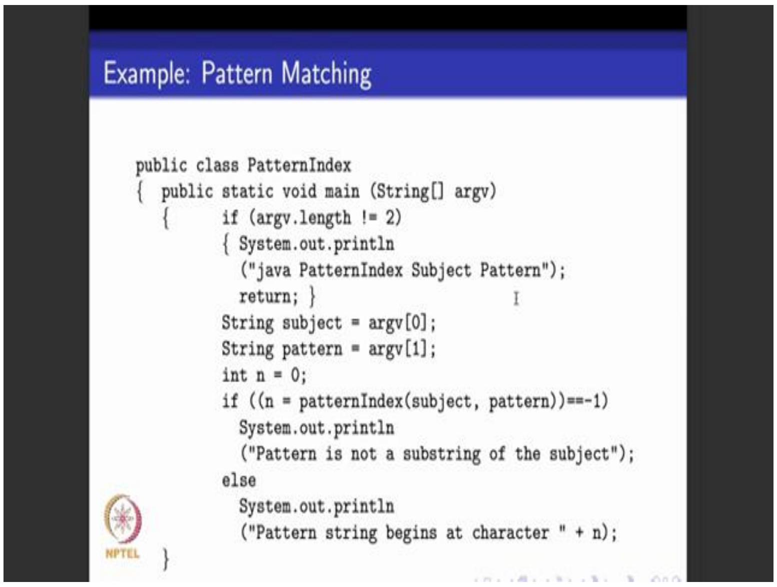

Data in graphs: Example

- Defs:
  - $def(1) = \{x\}$,
  - $def(5) = def(6) = \{z\}$.
- Uses:
  - $use(5) = \{x\}$,
  - $use(6) = \{x\}$.

So, here is a small example. Here is a graph that has 7 nodes, initial node is one final node is 7. Not all nodes have defs and uses only few nodes have. What I have a done here? I said at the node 1 there is a statement which says x is equal to 42, at node 5 there is another statement which says z is equal to x into 2, at node 6 there is another statement which says z is equal to x minus 5. So, you can think of this statement x is equal to 42 that comes at the initial node 1, as defining the value of x at node one right. So, we say definition one def of one is this singleton set {x}. 2 and 3 and 4 nothing happens, I move on, I look at 5. For statements at nodes 5 and 6 have expressions that calculate the value of z based on the value of x right. So, at 5 we have z is equal to x into 2, at 6 we have z is equal to x into 5.

So, nodes, these 2 nodes can be thought of as having statements that define z. So, we say definition of 5 and definition of 6 both are the singleton set {z}. Now how is z defined at nodes 5 and 6 z is defined using an expression that involves a constant and it involves the variable x. Similarly, here z is defined using another expression which involves this constant 5 and a variable x. So, we said x is used at nodes 5 and 6. So, is it clear? So, the first occurrence where the value of x is initialized or set to some value or declared or read from input is it is definition. In this case the def, x is defined at node one and whenever there is a statement that puts this defined value into use right like here at nodes 5 and 6, we say x is used in those places. In addition to that at nodes 5 and 6 the value z is assigned an expression involving a x, and so we say x is defined at nodes 5 and 6. We look at a more detailed example to understand this clearly.

(Refer Slide Time: 11:05)



Example: Pattern Matching

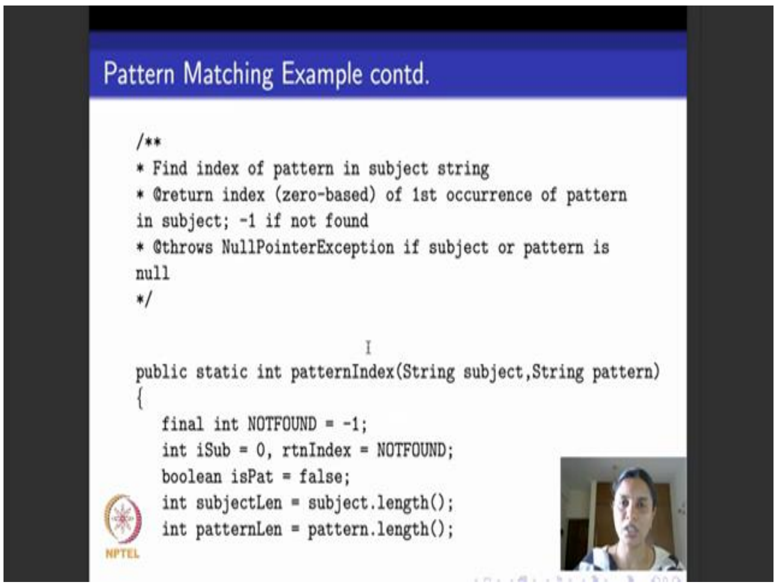
```
public class PatternIndex
{  public static void main (String[] argv)
   {      if (argv.length != 2)
          { System.out.println
            ("java PatternIndex Subject Pattern");
            return; }
          String subject = argv[0];
          String pattern = argv[1];
          int n = 0;
          if ((n = patternIndex(subject, pattern))==-1)
            System.out.println
            ("Pattern is not a substring of the subject");
          else
            System.out.println
            ("Pattern string begins at character " + n);
   }
```

So, here is an example that deals with the Java code corresponding to a pattern matching example. Some version of pattern matching or the other if you look at you will find it in several papers and books related to testing. It is one of the popular examples that we always deal with in testing because mainly because it is got lot of rich control flow structure. So, what I have done here is I have put a Java code for pattern matching that was taken from this text book, Introduction to Software Testing by Ammann and Offutt. What I have done is to make it readable, I have spread across this Java code over 3 slides. So, we first go through the program line by line, well understand what this program does, then well go ahead and draw the control flow graph of this program, and look at how the data or the variables that come in this program can decorate the control flow graph, where is it defined where is it used right. We use this example to understand the defs and uses of all the variables on the control flow graph corresponding to this code.

So, this is probably the first full piece of code that you are looking at and we will reuse this example let a few other places as we go down in our lectures. So, what does this code do? This code takes two arguments: a subject and a pattern, both of them are strings. And then it looks searches for a pattern in that subject. Like for example, I could say subject is a string which says institute, pattern could be something like ins right. If ins as a pattern comes in the subject, then you say that this pattern is found in the subject and you return index at which this pattern begins in the subject. If the pattern is not found in the subject then you return a string called -1. So, that is what it says. So, it says that you return -1. If the code returns -1 then your output saying pattern is not a substring of the subject. If the code returns any other character, then you say pattern does occur in the subject as a substring. And it begins at this particular index, yeah one more thing to note is this we look at a pattern as a contiguous substring we do not look at it as bits and pieces.

So, this is the first slide containing the first half of the code, which does on the initializations in the main prints .

(Refer Slide Time: 13:30)



Pattern Matching Example contd.

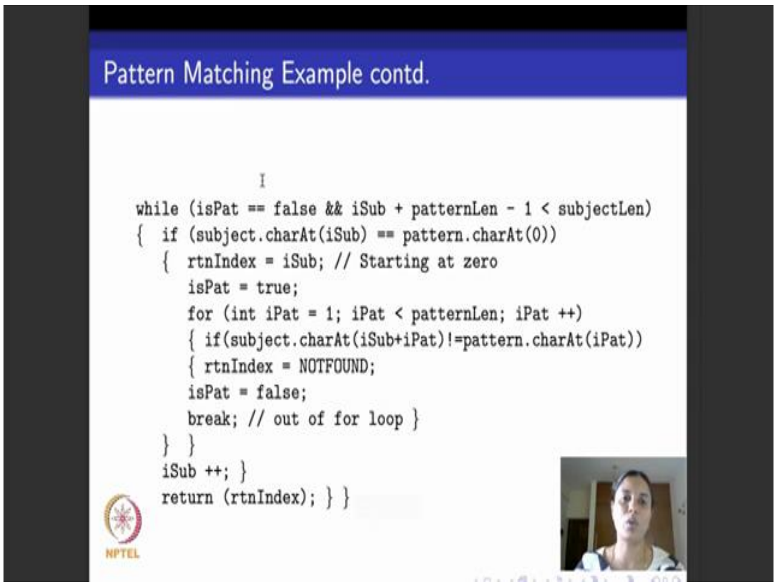
```
/**
* Find index of pattern in subject string
* @return index (zero-based) of 1st occurrence of pattern
in subject; -1 if not found
* @throws NullPointerException if subject or pattern is
null
*/

public static int patternIndex(String subject,String pattern)
{
   final int NOTFOUND = -1;
   int iSub = 0, rtnIndex = NOTFOUND;
   boolean isPat = false;
   int subjectLen = subject.length();
   int patternLen = pattern.length();
```

The actual code is given in continued across 2 more slides. So, here is the code that is continued from the first slide into the second slide. So, as I told you what is the goal of this code ? It is to find the index of the pattern, if it occurs in the subject and it will return the index 0 based in the sense that we count indices starting from 0, the first character of subject is index is 0, right. If the first occurrence of pattern in the subject we return -1 if it is not found if subject or pattern is empty then we threw a null pointer exception right. How does this pattern index work ? As I told you takes 2 arguments a subject and a pattern, and it does all these kinds of initialization?

So, it uses these variables not found, it initializes it to -1 in integer variable. It uses 2 other integer variables, isub you can read it as index that moves across the subject it initializes the index of the subject to 0. And then it sets the return index to be -1 which is not found. Remember when the return index continues to be -1 if you go back to the previous slide I return saying pattern is not a substring of the subject. And then it uses a few Boolean variables, ispat standing for 'is pattern', initializes it to false and then it calculates the length of the subject and the length of the pattern. Why do we need the length of the subject and length of the pattern ? Because we want to use a for loop to move across the length of the subject from left to right and another for loop to move across the length of the pattern from left to right?

(Refer Slide Time: 15:08)



Pattern Matching Example contd.

```
while (isPat == false && iSub + patternLen - 1 < subjectLen)
{  if (subject.charAt(iSub) == pattern.charAt(0))
   {  rtnIndex = iSub; // Starting at zero
      isPat = true;
      for (int iPat = 1; iPat < patternLen; iPat ++)
      { if(subject.charAt(iSub+iPat)!=pattern.charAt(iPat))
      { rtnIndex = NOTFOUND;
      isPat = false;
      break; // out of for loop }
   }  }
   iSub ++; }
   return (rtnIndex); } }
```

So, here is the main part of the code. There is a while loop inside this there is a condition there is a for loop, and then there is an if loop. Why am I talking about this while if for if, is to tell you that this code has very rich control flow structure. The control flow graph of this code will involve some amount of branching which is a very which makes it a very interesting case for testing and here there are lots of nested loops one inside the other. So, the control flow structure it becomes even richer and so it is an interesting example to look at for testing. Now going back and understanding what the code is. So, it says as long as it 'is pattern' turns out to be false and pattern index is within the subject of the length of the pattern is within the subject index then what do I do in this if loop, I start matching character by character.

So, I say wherever I am in the subject which is this index of subject isub, does it match the first character of the pattern? First character of the pattern is present at index 0. If it is then, you say that the return index could be this value. Let us say if you take the earlier example that we had right, let us say a subject was the string institute and the pattern was ins, so, right up front at the index 0 itself, the subject and the pattern match, i and i match right. So, you set the index where it matches to be the return index and then you set this Boolean flag is pattern found which is ispat to be true right.

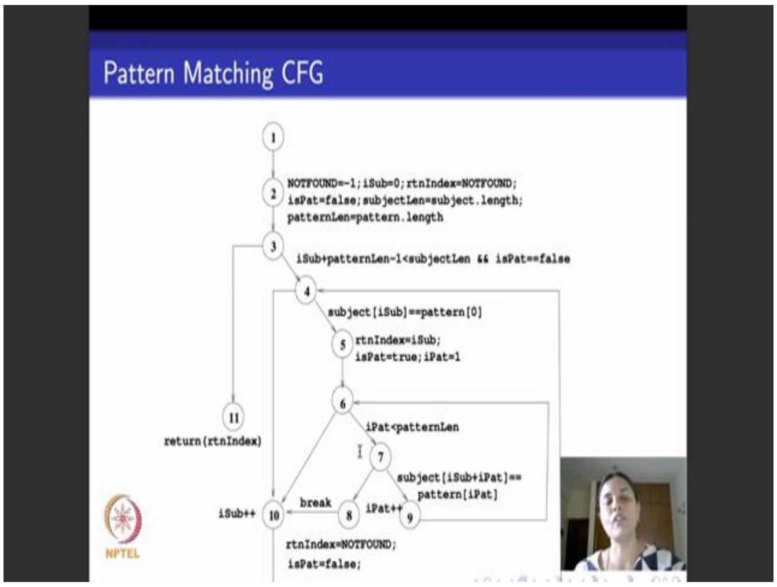
Now what you do you enter a for loop, keep incrementing the position that you look at in the pattern as long as you go till the end of the pattern and see if every character of the pattern, one at a time, matches the corresponding character in the subject that is what this if statement does. If there is a mismatch at some point in time it says he sets the return index to not found and makes is pattern false and comes out of the for loop right. If every character of the pattern continues to match the subject it keeps incrementing the index in the subject successfully finishes the for loop and it will return this value here that it is set as the return index. So, is it clear what the code does?

So, I just to quickly recap we are looking at this pattern matching code. It takes 2 arguments, a subject and a pattern and it uses a few Boolean variables to search for the pattern in the subject. What it does is initially it assumes that the pattern is not found in the subject. So, it says 'is pattern' is false, and then here it checks with the length of the pattern is well within the length of the subject and then it says now let us look at the first character of the pattern which is at index 0 along with wherever I am in the subject which is given by this index isub, if they match if this return statements gives true, then you say I may have found the pattern beginning at this index.

So, you set that to be the return index. And then you say I have found the pattern set you to true, then you enter a for loop where you continuously walk down the pattern and check if every character in the pattern that you encounter, in sequence, matches the subject from the index in the same sequence. If there is a mismatch at any point in time then you reset a return index to not found say pattern is not found come out of the for loop. But if we successfully finish executing this for loop then you increment the subject and then you can return this value of the index where the pattern was found in the subject.

So, now our first goal is to draw the control flow graph corresponding to this program right. So, as I told you for control flow graph this program is spread across these 3 slides, but I am ignoring this part of the code, and this part of the code which is just commented out. I begin to draw control flow graph mainly from here, from this main pattern index method. So, we see this pattern index method it has a whole series of assignment statements, it has a loop, a branch, another loop another branch.

(Refer Slide Time: 19:15)



Pattern Matching CFG

1
2
NOTFOUND=-1;iSub=0;rtnIndex=NOTFOUND;
isPat=false;subjectLen=subject.length;
patternLen=pattern.length
3
iSub+patternLen-1<subjectLen && isPat==false
4
subject[iSub]==pattern[0]
5
rtnIndex=iSub;
isPat=true;iPat=1
6
iPat<patternLen
7
subject[iSub+iPat]==
pattern[iPat]
11
return(rtnIndex)
iSub++
10
break
8
iPat++
9
rtnIndex=NOTFOUND;
isPat=false;
NPTEL

So, here is how the control flow graph of that main method looks like. This node one is a dummy node which represents this statement. It says subject and pattern are passed to this node. There is no concrete statement representing this so I have left it blank. Node 2 stands for all these initializations that you do in the beginning of the program right all these statements these 6 statements that are there. You could put it in different nodes, but typically when we draw control flow graph, when we have a sequence of assignment statements without any control statement in between just assignment statements one after the other, it is a standard practice to collapse them all into one node and put it as one node and then augment that node with all these statements. That is what I have done because there is not much point in calling each of them as separate node, we really do not find any use for it when it comes to testing.

So, we collapse it all and put it into one node. So, this single node represents all these assignment statements. After that what do we do with that code we get into this while loop right. So, that is this while loop at node 3, I say this is the condition that I check in the while loop you can see that is the condition I check. This condition could be true this condition could be false. Just for increasing the readability I have written it only as a label for this node the reverse of this condition the negation of this predicate is true here. I have not mentioned it because the graph was becoming the figure was becoming too cluttered. So, I have just left it at that right. So, read the label of this edge as negation of this.

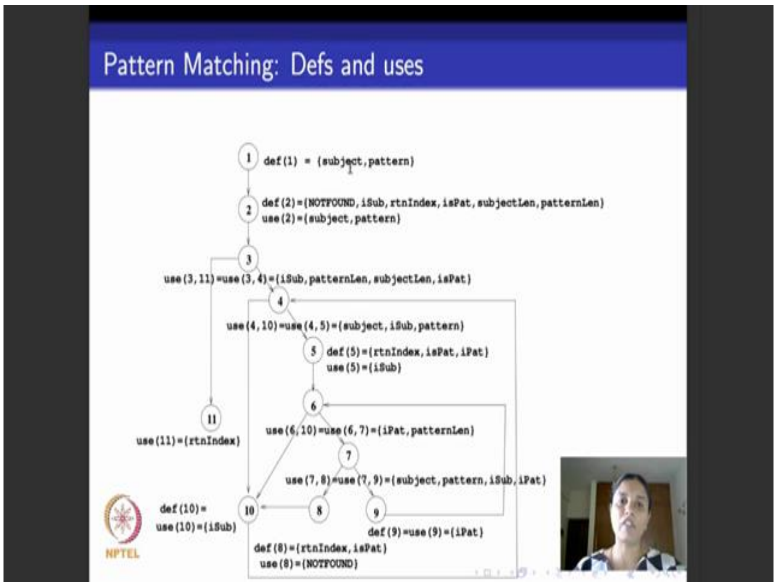
So, I am entering my while loop here. As per the code the next is an if statement which check for another condition. So, that I am entering my if statement here, if statement returns true, this is if statement returning false. After my if statement returns 2, I do two assignments I set, what does is it mean for the if statement to return true I have found the first match. First character with the pattern has found a match in the subject. So, I do these 2 assignments if you remember right I set the index and say I have found the first match by setting the flag to be true. So, that is this node 5.

After that I enter, sorry, after that I enter this for loop, that is this. This is the statement corresponding to the for loop being true. And inside the loop I keep checking if the pattern index matches the subject index. At node 9, I increment the pattern index go back and repeat this for loop here from 6 to 9 and when the condition fails I come out with a for loop. I break out of the for loop and I am here that is what this represents and then this is the thing that they use to go back for the main while loop. This is when I exit the main while loop, I do this final statement which is return index. Is it clear? I am sorry I have to go back and forth across the slides because there is no way I could have put the whole a program and the control flow graph into one slide.

So, just to summarize we have looked at a pattern matching program that I had presented to you across 3 slides which were this, this and this. It is a program that has rich control flow structure looks if pattern given as a string comes with a substring of a subject. What we did here was to take the main method in the program and draw it is control flow graph.

So, I will repeat this exercise of how control flow graph will look like for various program constructs, but hopefully this example will help you to understand it also. Now we go back to the main goal of this lecture which is to understand data right, data definition and data use. What is data for this particular program? Data is all these variables: NOTFOUND, isub, rtnIndex, subject, pattern, ispat. So, many variables are there right. So, what do I do, I take the same control flow graph instead of annotating it with statements I say which are the variables that are defined and which are the variables that are used at various nodes in the graph and various edges in the graph.

(Refer Slide Time: 23:46)


Pattern Matching: Defs and uses
1
def(1) = {subject,pattern}
2
def(2)={NOTFOUND,iSub,rtnIndex,isPat,subjectLen,patternLen}
use(2)={subject,pattern}
3
use(3,11)=use(3,4)={iSub,patternLen,subjectLen,isPat}
4
use(4,10)=use(4,5)={subject,iSub,pattern}
5
def(5)={rtnIndex,isPat,iPat}
use(5)={iSub}
6
use(6,10)=use(6,7)={iPat,patternLen}
7
use(7,8)=use(7,9)={subject,pattern,iSub,iPat}
8
9
def(9)=use(9)={iPat}
def(8)={rtnIndex,isPat}
use(8)={NOTFOUND}
10
def(10)=
use(10)={iSub}
11
use(11)={rtnIndex}
NPTEL

So, that is this graph that I have drawn here. It is a same control flow graph instead of depicting statements it now depicts definitions and uses. So, if you look at node number one, it says the definition of one that is at node number one, which are the two variables that are defined? The Two variables that are defined at one are subject and pattern. You might wonder this here it is blank and while have I suddenly put def(1) as subject and pattern as I told you when you explain the CFG read this node one as these two patterns subject and pattern being passed on as parameters to this method. Because this passed as parameters I am not depicted here, but because it is passed as parameters it is defined here.

Now, what is the def (2)? If you see there are, so many assignment statements here which is not found as -1, isub is 0, rtnIndex is NOTFOUND, ispat is false and so on right. So, I say definition of 2 is all the variables that have been initialized here, which is NOTFOUND, isub, rtnIndex, ispat, subject length, pattern length. Now 2 also has two uses which is subject and pattern. Why is it use of subject and pattern I use the variable subject and pattern that was passed as parameters at node 1 and compute their length here subject length and pattern length and assign it to subjectLen and patternLen. So, I have used these 2 variables subjectLen and patternLen. So, that is why I have put use of 2 as subject and pattern right. Is one more settle point do not worry about it. If you do not understand it if you see I have initialized not found to be minus 1. So, I have to def(2) as not found and I have also initialize a return index to be not found.

So, strictly speaking I should put use(2) as not found. So, you could go ahead and do that, but at the little later well see that we typically do not consider what are call local uses of a variable. Because I have collapsed all these 5, 6 assignment statements into one node in the control flow graph, I say this variable is defined and used within one node. So, it is like a local use of a variable. We typically do not capture local uses of variables, but if you are not comfortable with this if you want to see use(2) it is subject pattern and not found you can go ahead and do it there is no harm in it right like this I move on.

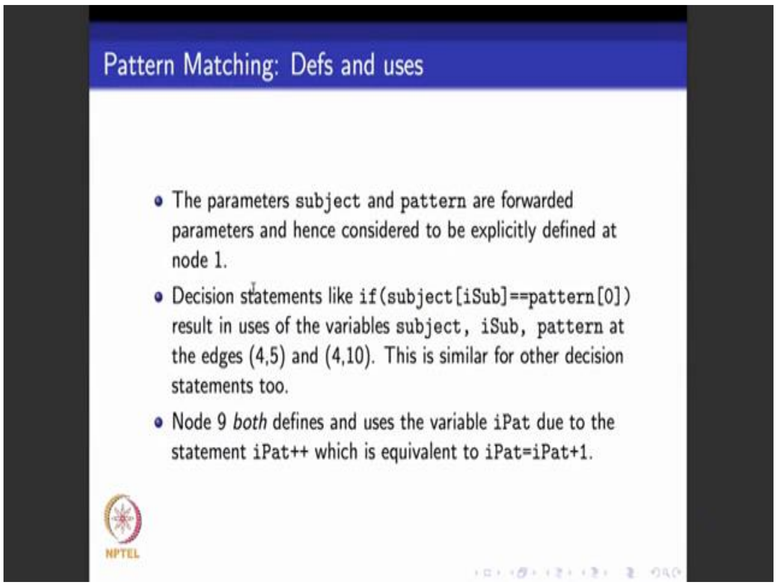
Now, I can define defs and uses for edges also remember well saw definition and uses for vertices and definition and uses for edges. So, which is this, if a predicate that labels this edge. So, it uses thus a variables isub, patternLen, subjectLen and ispat. So, that is what I have written here. The use of this edge is this predicate i is this set isub pattern length subject length ispat. If you remember I told you that this corresponds to a branch, this is the branch where I enter the while loop. So, this predicate is true. This is the branch where I exit the while loop where the predicate becomes false.

Just for readability I didnt write the condition here, but the same set of variables with this condition negated comes here also. So, I say use of this edge (3, 11) and this edge (3, 4) is the same set right. Because the both the predicates one the positive version and one the negated version both will use the same set of variables. Similarly, this is the if statement. So, use of this edge (4, 10) and this edge (4, 5) is this set subject, isub, pattern and then at statement 5, I have these 2 assignment statements. So, I say rtnIndex, ispat and ipat are defined at the statement. Please note that ipat is defined once here also sorry, ispat is defined once here also at 2 and it is defined at 5.

So, if you go back and look at the code it is initialized at 2 and its value is reset to another value at 5. So, both we call them as definitions because it is again written, rewritten in memory. So, definition of node 5 is this set. Similarly use on node 5 is this isub, is this isub, is it clear. So, if I move on like this. Use of this edge (6, 10) and this edge (6, 7) are all the variables that come in the predicate corresponding to the for loop, use of this edge (7, 8) and the edge (7, 9) are all the predicates that come in the if loop. And then what do I do at 9 at 9? I do this, I increment the index of searching in the pattern by using ipat+. So, what is ipat++, that reads as ipat is equal to ipat plus 1. So, if I read it that way I am using ipat both in the left hand side and in the right hand side. So, it is both defined and used at variable 9.

So, that is where def of 9 is the same as use of 9 and that is the single concept ipat right I go on like this finish right. So, what do I do? Just to recap, we have a program like this. We define the control flow graph corresponding to the program, from the control flow graph I can take the basic control flow structure and instead of augmenting with statements I augment it with defs and uses of all the variables. So, is it clear how to do it? So, before we move on I will go back to the slide where we define defs and uses. So, what is a def, is a place wherever the value of the variable is stored into memory. What is a use, a place where value of the variable is accessed in a memory right? Typically, we take a graph and for every node in the graph, we call what is def of n and for every edge in the graph we say what is use of n and for every edge in the graph we say what is use of e and def of e.

(Refer Slide Time: 30:12)

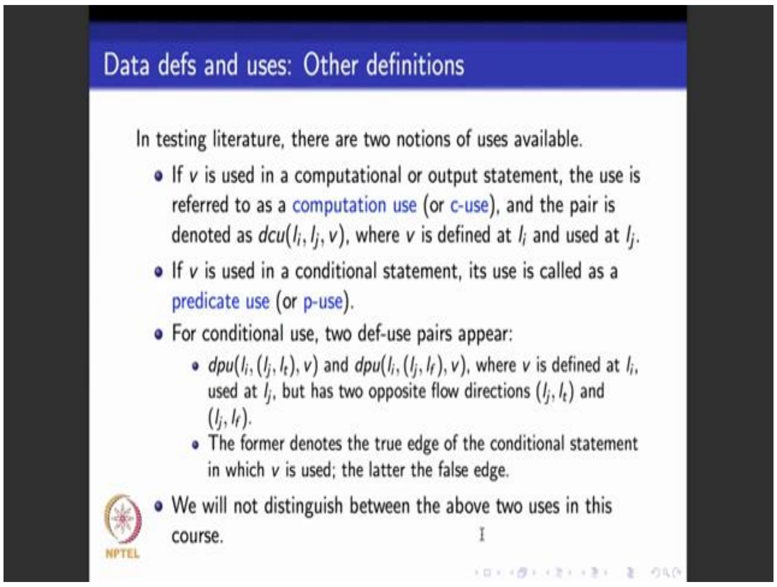


Pattern Matching: Defs and uses

- The parameters subject and pattern are forwarded parameters and hence considered to be explicitly defined at node 1.
- Decision statements like if(subject[iSub]==pattern[0]) result in uses of the variables subject, iSub, pattern at the edges (4,5) and (4,10). This is similar for other decision statements too.
- Node 9 *both* defines and uses the variable iPat due to the statement iPat++ which is equivalent to iPat=iPat+1.

So, this I think I explained this slide already what I said was at node one parameters subjected pattern of a forwarded parameter. So, they consider to be explicitly defined at node one. And I also explained the how these uses occur in decision statements that is what this line item says. And the node 9 has a statement which is like ispat++. Suppose I read it as ipat++ as ipat plus 1 then it can be thought of as both defining and using this variable right. So, that is why I write def of 9 is the same as use of 9 is the same as ipat.

(Refer Slide Time: 30:50)



Data defs and uses: Other definitions

In testing literature, there are two notions of uses available.

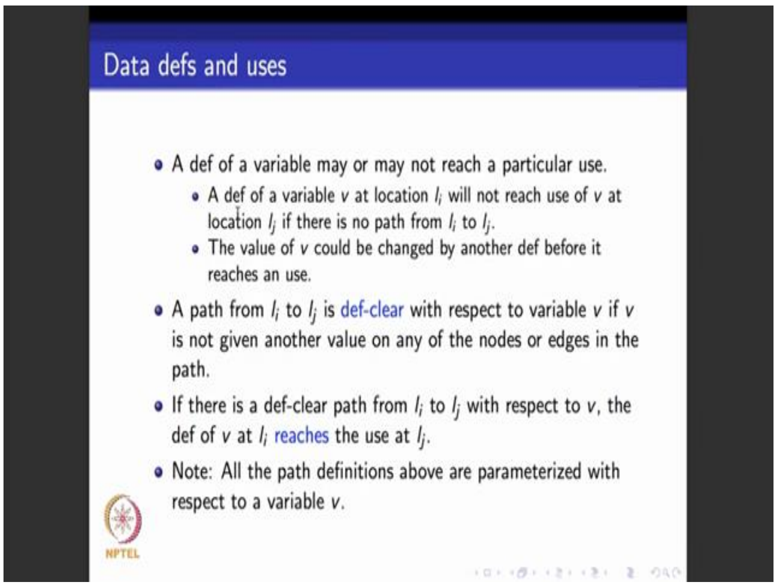
- If v is used in a computational or output statement, the use is referred to as a computation use (or c-use), and the pair is denoted as $dcu(l_i, l_j, v)$, where v is defined at l_i and used at l_j.
- If v is used in a conditional statement, its use is called as a predicate use (or p-use).
- For conditional use, two def-use pairs appear:
  - $dpu(l_i, (l_j, l_t), v)$ and $dpu(l_i, (l_j, l_f), v)$, where v is defined at l_i, used at l_j, but has two opposite flow directions (l_j, l_t) and (l_j, l_f).
  - The former denotes the true edge of the conditional statement in which v is used; the latter the false edge.
- We will not distinguish between the above two uses in this course.

Before we move on, traditionally in literature there is a paper by Wyeuker and a a student by name Sarah, which is well quoted for data flow graphs. They distinguish between two different kinds of uses. So, they say when a use occurs as a part of an assignment statement or an output statement that is what is called a computation use or a c-use. And a use could occur as a part of a condition statement right like we saw here this use is a part of a definition at node 2. Whereas, this use at this edge is a part of a predicate or a condition that occurs as a part of the if statement.

So, they distinguish between these 2 uses, one is a computation use or use that occurs as a part of a printf or as an assignment statement is what is called a computation use. A use that occurs as a part of a decision statement is what is called a predicate use. So, instead of talking about du-pairs definition use pairs they talk about definition computation use d c u pairs or definition predicate use d p u pairs. So, when I talk about a definition computation use d c u pair I say a variable v is defined at l i and is being subject to computation use at l j that is it is used as a part of an assignment statement or as a printf statement at l j. When I talk about d p u I say the variable v is defined at statement l i it is used in the predicate l j and l t says that this predicate at location l j has evaluated to true and here l f says the predicate at a location j has evaluated to false.

So, when I use it on as a part of a predicate which could be as a part of an if statement or a while statement, the single purpose of that use is to make the predicate evaluate to true or evaluate to false. So, and based on whether it evaluates to true or false, the paths that it takes in the control flow graph could be different. So, this two paths are distinguished. This is just for your information for our purposes of these lectures and this course we will not distinguish between computation use and predicate use. We will just say all of them are uses. That is what I have done in this control flow graph right. If you have to be the strictly correct you could call this is predicate use you could call this is computation use, but for the purposes of this course and this lecture everything would just be use.

(Refer Slide Time: 33:32)



Data defs and uses

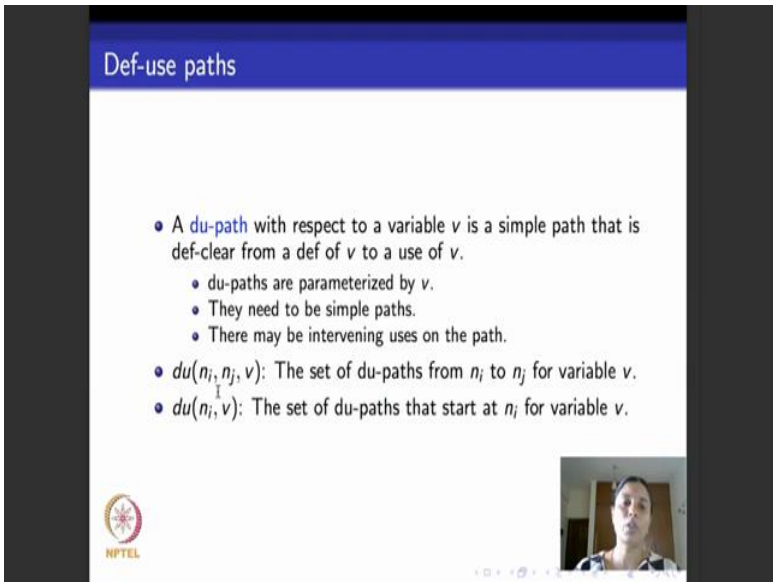
- A def of a variable may or may not reach a particular use.
  - A def of a variable v at location l_i will not reach use of v at location l_j if there is no path from l_i to l_j.
  - The value of v could be changed by another def before it reaches an use.
- A path from l_i to l_j is def-clear with respect to variable v if v is not given another value on any of the nodes or edges in the path.
- If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i reaches the use at l_j.
- Note: All the path definitions above are parameterized with respect to a variable v.

What will be the concerns about definitions and uses? As I told you in the beginning of this lecture a definition may or may not reach a particular use. Why will it not reach a particular use? It could happen, it could have happened that the clear a variable was defined at a particular location and used at some other location l j. And then there was no path from l i to l j, l j could be a part of a dead code in the program or it could be the case that the value was defined and before it was used it was defined again. It was initialized and before it was put to use again maybe it was defined by some other statement once again right. So, such things can happen.

So, we talk about several kinds of paths in the control flow graph. So, we say a path from a location l i to another location l j is def-clear with respect to a variable v if v is not given any other value on any of the nodes or edges in the path. So, I take a control flow graph augmented with definitions and uses take a location l j, l i take another location l j v is defined at l I, a variable v is defined at l i.

Now, there is a path from l i all the way to l j which could be another node or an edge. I walk along the path. In between in the path if the variable v is not defined once again then you say such a path is def-clear. If there is a def-clear path from l i to l j with reference to a variable v, we say that the definition of v at l i reaches the use at l j. As I told you whenever we talk about definitions and use this always implicitly a parameter of a variable that is involved.

(Refer Slide Time: 35:17)



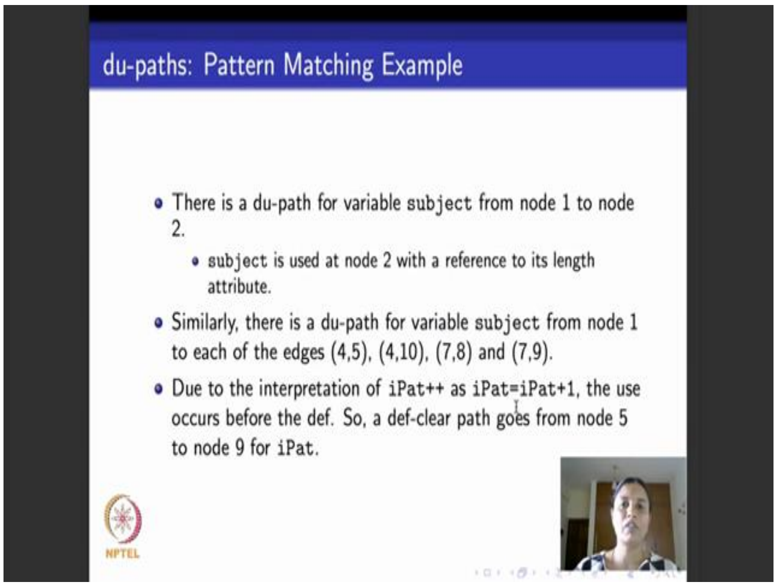
Def-use paths

- A du-path with respect to a variable v is a simple path that is def-clear from a def of v to a use of v.
  - du-paths are parameterized by v.
  - They need to be simple paths.
  - There may be intervening uses on the path.
- $du(n_i, n_j, v)$: The set of du-paths from n_i to n_j for variable v.
- $du(n_i, v)$: The set of du-paths that start at n_i for variable v.

So, now, we talk about what is called a def use path or a du-path. What is a du-path? A du-path with respect to a variable v is a simple path that is def clear from a definition of v to a use of v right? So, what it says in simple terms is that take a node or an edge, where a variable is defined in our case it will be a node a node where a variable is defined keep going down and take it is first use right. In between assume that the variable is not defined and the only other condition that we insist is that this path be simple, there are no cycles in this path. If such is the case then we say that I have found a path from a definition of a variable to a use of a variable and I abbreviate it and call it as a du-path right. So, du-paths have to be simple paths and they have to go from a definition to a use. In between there could be intervening uses, but no definitions.

So, here is a a notation for that, you say du (ni, nj, v) means it is the set of all definition use path from location n i to location n j for the variable v. What is (du, ni, v)? It is a set of all d u paths that start at n i for a variable v.

(Refer Slide Time: 36:38)



du-paths: Pattern Matching Example

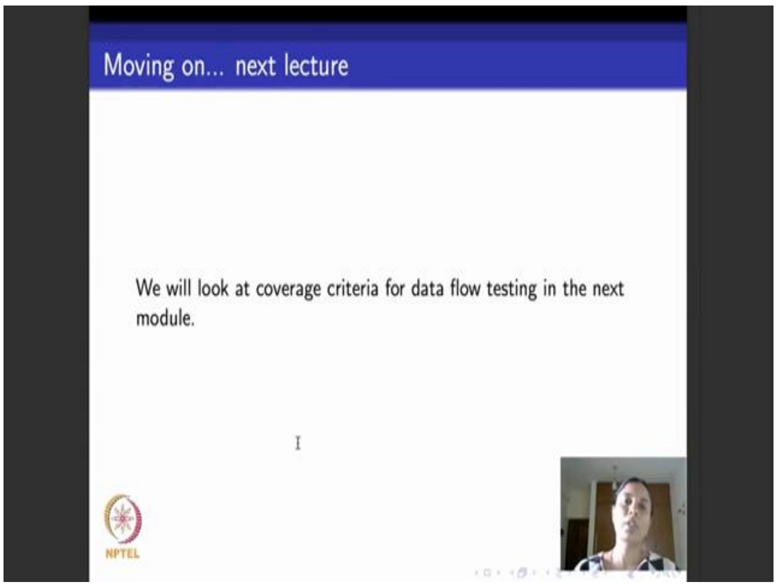
- There is a du-path for variable `subject` from node 1 to node 2.
  - `subject` is used at node 2 with a reference to its length attribute.
- Similarly, there is a du-path for variable `subject` from node 1 to each of the edges (4,5), (4,10), (7,8) and (7,9).
- Due to the interpretation of `iPat++` as `iPat=iPat+1`, the use occurs before the def. So, a def-clear path goes from node 5 to node 9 for `iPat`.

If we go back and trace out what are the d u paths. So, if you see there is a small d u path here. There is a definition of subject here and then there is a use of subject here. So, I can say if there is a definition use path for the variable subject. So, there is a du-path. Similarly the term subject is defined here and look at other nodes where it is used. Like for example, if you see it is used in this edge (4, 10) and (4, 5) it is you would comes as use of (4, 10) and (4, 5) and in between it is not defined again.

So, this path from 1, 2, 3, 4 to the edge (4, 5) or to the edge (4, 10) is a du-path for the variable subject. So, hopefully it is clear what I am saying. There is a du-path for the variable subject from the node 1 to the edges (4, 5), (4, 10). Similarly, if you go back a there is a du-path for the variable subject from the node 1 to (7, 9) and (7, 8), because it is being used here also, but not defined anywhere in between. Please remember a du-path is a path between a definition and a use such that there are no definitions in between in the path of that variable. There could be uses in the path of that variable, that is all right. Like for example, if I take the du-paths from 1, node 1 to the edges (7, 8) or (7, 9), there is a use of the variable subject in between we allow that, but we say in between a du-path in intermediate nodes or edges there should not be another definition, a du-path with respect to a variable v is a simple path that is def clear from definition to a use, but there could be intervening uses on the path.

So, another example of a du-path for that pattern matching thing is this ipat++, which was if you remember at node number 9, if I read ipat++ as this statement then I can interpret it as the use occurring before the def because this right hand side needs to be executed before the left hand side. So, I with that interpretation I can say that there is a def clear path that goes from node 5 to node 9 for the variable ipat.

(Refer Slide Time: 38:59)



Moving on... next lecture

We will look at coverage criteria for data flow testing in the next module.

So, that is all I have to tell you for today's lecture. What we did today just to summarize is assuming that we have a model of a software artifact typically a code as a graph, which means as a control flow graph how I actually look at the variables that come in the in code, and decorate the control flow graph with the definitions and uses of a variable. So, once I do I understand when a variable is defined and when a variable is used, it could be defined and used at nodes. It is typically used only at edges not defined at edges, then I talk about a du-path or a def clear path and so on. What we will do in the next lecture is to see how these du-paths can be used to define various coverage criteria that will help us to understand, when a variable is defined is it being used, is it being not used and so on and so forth. So, we look at data flow coverage criteria based on definitions and uses in the next lecture.

Thank you.