Artificial Intelligence:

Prolog: Controlling Search

Prof. Deepak Khemani

Department of Computer Science and Engineering

Indian Institute of Technology, Madras

Module – 06

Lecture - 05

Module 6 Lecture 5

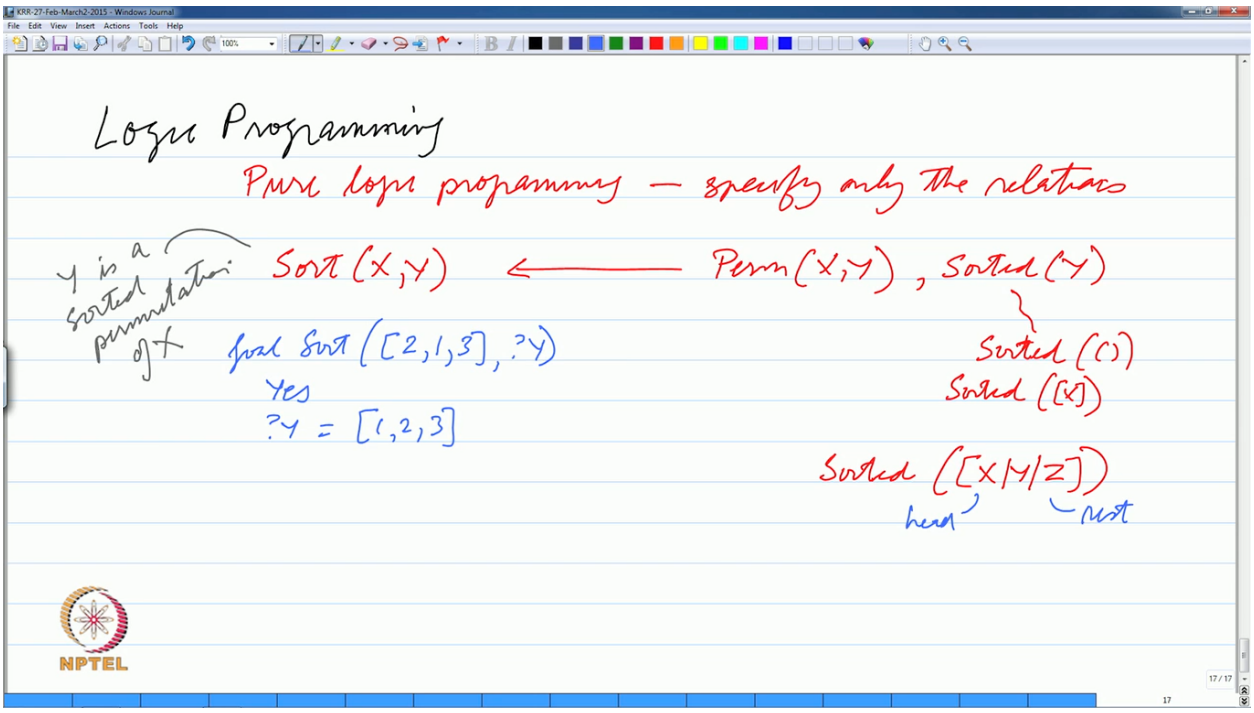
Okay so let's continue with our study of logic programming and we had mentioned briefly that Prolog is an implementation of what we can now say is an approximation to logic programming. In pure logic programming you have to specify only the relation and the rest can be left to inference engine which can somehow figure out which rules to apply to reach facts and so on. So as an example of what can be a pure logic program which basically focuses on the relation we can try to define a sort a program to sort lets say numbers. So we can say that Sort X Y by this we mean Y is a sorted permutation of X. the semantics of Sort X Y is that we have two lists one of them is X and one of them is Y and Y is a sorted permutation of X. so if you wanted to run this program you would say Goal for example Sort 2 1 3 Y and it would say Yes if Y is equal to 1 2 3 for example if we are sorting in ascending order.

So you could give that the first argument is unsorted array and the second argument which the program will find for you is the sorted array. We can define in a very simple fashion by saying by just rewriting the definition I have written in as a logic program. We can say that find a permutation Y of X and test whether its sorted or not. so this is trying to get at the basic idea of why did people think of logic programming. Essentially their goal was that as a programmer or as a user you should be able to focus on the relations between input and output and not so much on the procedural control of what is the next operation that I must carry out and things.

So you should focus on specifying the relation. If you want to write a program to sort you should say okay what do you mean by sorting. You mean by sorting that you find a permutation which is sorted then you have to ofcourse define what do you mean by sorted and so on. So how do you define sorted for example. You could say Sorted an empty list is sorted. Then you could say let me use this square bracket notation. You could say that a list with one element is sorted by definition but if it's a longer list and I will use this notation, let me use the old notation or may be this one was better.

This notation basically is a short form for Cons notation so X is the head of the list, Y is the second element of the list and Z is the rest of the list. So the rightmost element is the rest of the list. And this is the head and recursively you can say that between and Y and Z, Y is the head and Z is the rest.

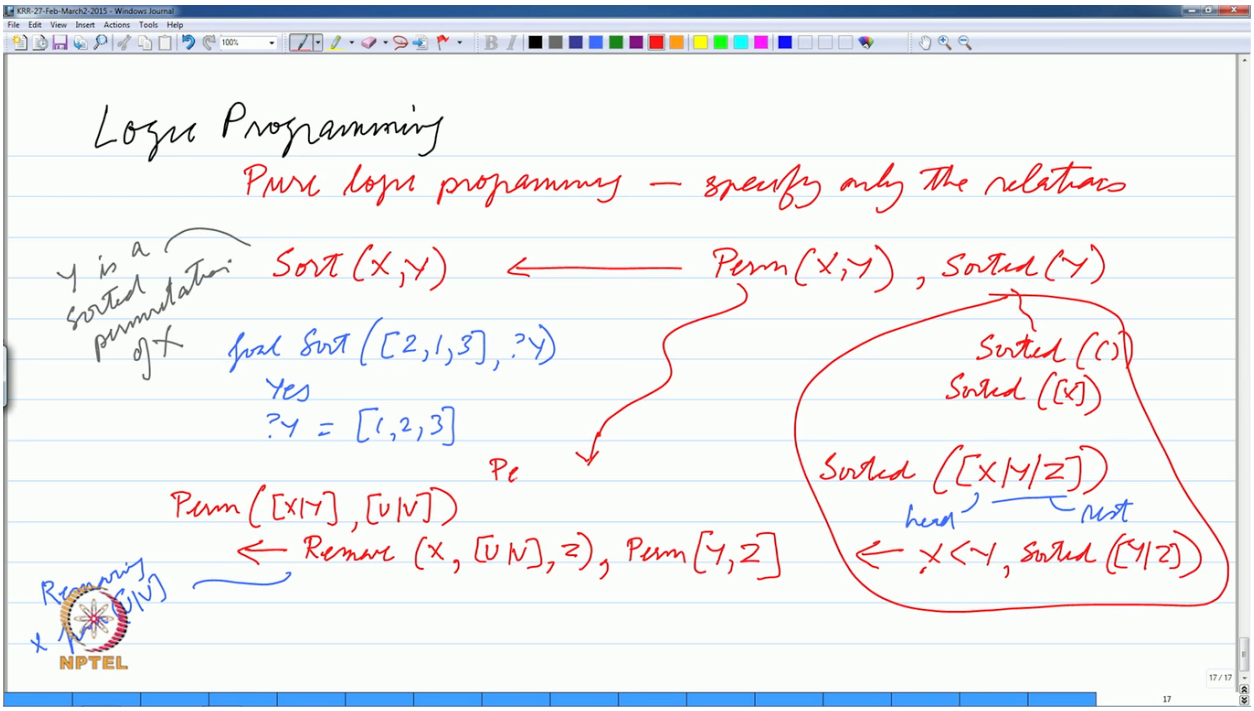
(Refer Slide Time: 6:05)



So initially this is the rest and so on so forth. So the basic idea is that you can define what you mean by sorted by saying that a list which has atleast two elements is sorted if X is less than Y. lets say we have this ordering between elements defined already. X is less than Y and Sorted rest of the list which comprises of Y comma Z which recursively will look at the next two elements. So essentially what you are saying is that a list is sorted if the first element is less than the second element and the rest of the list is sorted recursively which again will look at the second and the third element and see that the second is less than the third element. Then third is less than the fourth element and eventually it will boil down to one of the two base clauses and program will terminate.

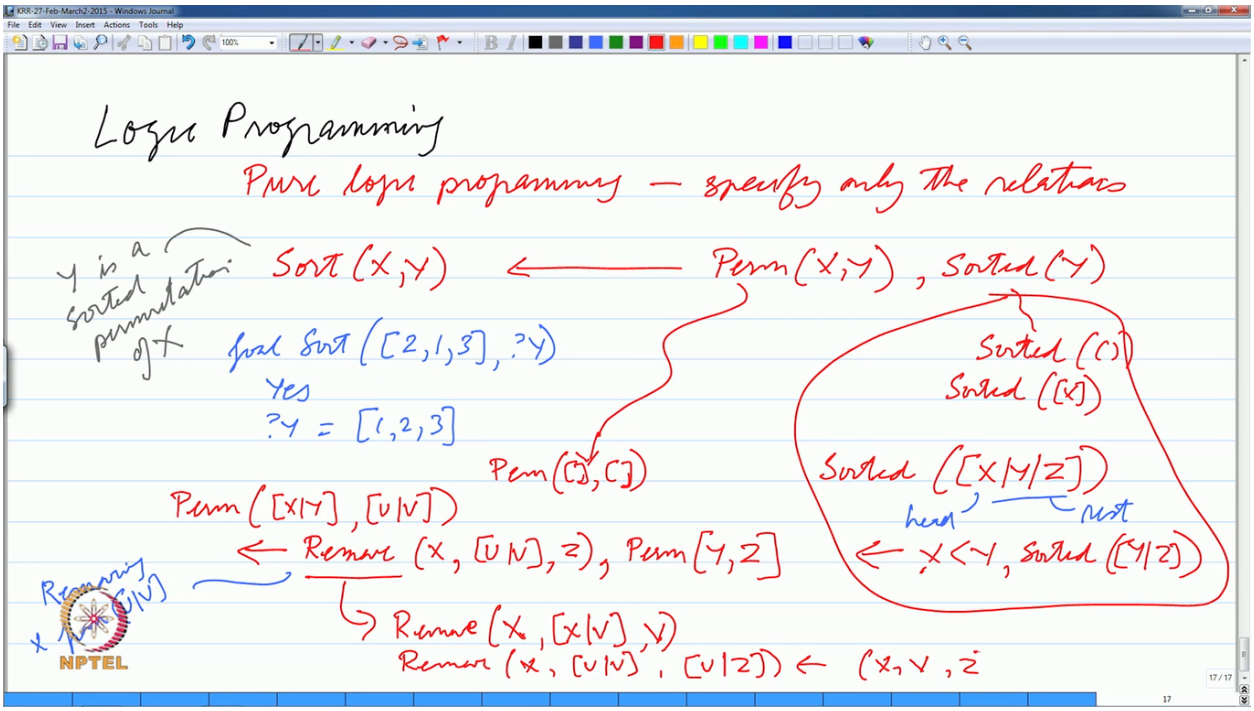
The first clause is just for the sake of completeness we are looking at empty list also. So we have defined lets assume that we can define what is sorted. What about defining permutation so you need to define that as well. So you could define permutation by saying something like that a list which starts with X and the rest is Y is a permutation of a list which starts with V and goes to U provided I write a predicate called Remove X from this list U V Z and the meaning of this is if I remove X from U V so this element X must be present somewhere in U V if I remove it from U V then I get Z and Y is a permutation of Z. so I am not writing the base clause you can write the base clause. Which says that empty list is a permutation of itself just write that.

(Refer Slide Time: 9:18)



That's the base clause. This clause which defines the permutation says that if I have two list, one of them is beginning with X and other one is beginning with U and V something then if I remove from the second list X and whatever remains I call as Z then that Z must be a permutation of Y. so I removed the first element from the I am saying that U V is a permutation of X Y if I remove X from U V and whatever remains is a permutation of Y. so ofcourse this X will be somewhere inside U V and this Remove itself you can define by saying Remove if it happens to be the first element then just return the rest of the list so that's the base clause for Remove. And the recursive clause would be that look inside V for X Remove X comma U V will give me a list which I can call as U comma Z provided X from V to give Z.

(Refer Slide Time: 11:45)



So this remove is also recursive program base clause says that if it's at the head you can simply remove it. If its not at the head then you should look inside the list to remove. So in any case what we have seen here is that you can write a logic program to sort the list and sorting as we know is an activity which is one of the most common activity in the industry. That most of the time program in industry are spent on sorting. Now if you were to use this program that we just wrote you could translate into prolog in quite straight forward manner but you can see that it's as inefficient a program as it can get because what is it saying. Its saying look for a permutation of the list and see whether it is sorted. Now obviously it doesn't make sense at all because the number of permutations of a list are very many.

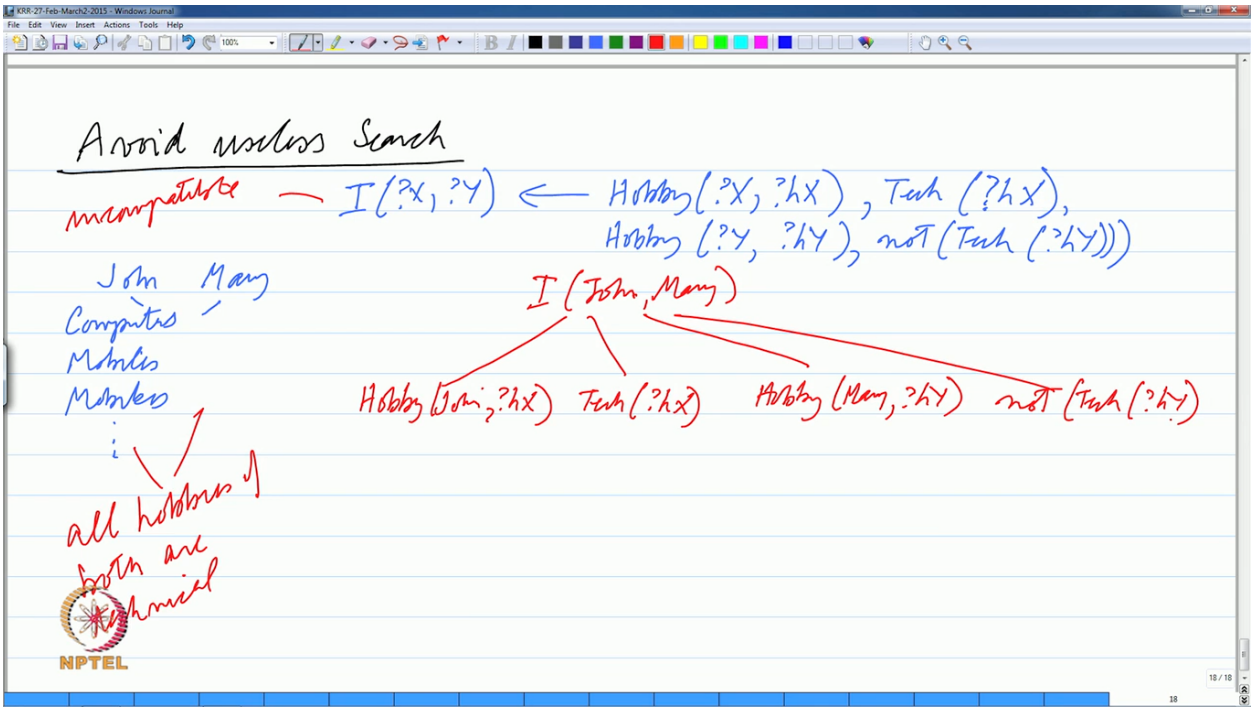
And it has its own way of prodding through all the permutations and then doing it. so obviously one doesn't want to write pure logic programs. One wants to write programs in prolog which are efficient. As an exercise I will ask you to write merge sort and quick sort in logic programming and you can see that all you have to do is define what is the merge sort. Basically a merge sort X if you say that Y is the merge sorted version of X and you are going to say that I am going to break up X into two parts lets call them X1 and X2. That you have to define logically what do you mean by breaking up. And ofcourse most of the time we say left half of the array and right half of the array but it doesn't really matter. You could simply say I will take the alternate elements, I will take the first element and put it in X1, second element in X2, the third element in X1 again, fourth element in X2. You could just separate the elements in any manner, break it down into separate problems. Recursively merge sort the two arrays X1 and X2 and then append the two outputs to give you the sorted. So you can write that as a logic program and I will leave this as a small exercise.

I want to focus on some other things which is again to do with efficiency and to avoid useless search. How can we since we are working with deterministic strategy we are trying to optimize our program so that they will run well in prolog. Lets see how you can avoid some useless search. So lets again work with an example to illustrate the idea. So we will say that two people are incompatible which I will just use I X Y . I keep forgetting to write the question mark so may be we should consistently write that. But I hope you would be able to make sense of it. if lets say Hobby X hX and Technical hX this is an example I have taken from Chamiak and McDemott.

So first to understand this definition what this definition is saying is that two people X and Y and incompatible if X has a hobby which is technical. Lets say by technical we mean having interest in mobile phones computers and motorcycles and things like that. And if Y has a hobby which is non technical so lets say non technical means lets say watching movie or listening to music or something like that it doesn't matter. Lets say we have only two kinds of hobbies, one is technical kind and one is non technical kind. And this definition is saying that two people are incompatible if one of them have technical hobby and the other one has non technical hobby. Now lets say we are talking about John and Mary.

And let us for the argument sake that both of them have the same hobbies computers mobile bikes what else is a technical hobby can you just tell me. Okay whatever lets say there are some set of technical hobbies. Now obviously if you were to ask a query Are John and Mary incompatible? The answer should be no. so essentially what we are saying is all hobbies of both are technical. So I am not listing them out. So if we ask this query that are john and mary incompatible we have to show that, list the find the hobby for john whatever the hobby is. Show that that hobby is technical. Find a hobby for mary and show that it is not technical. We will at some point see how to define not.

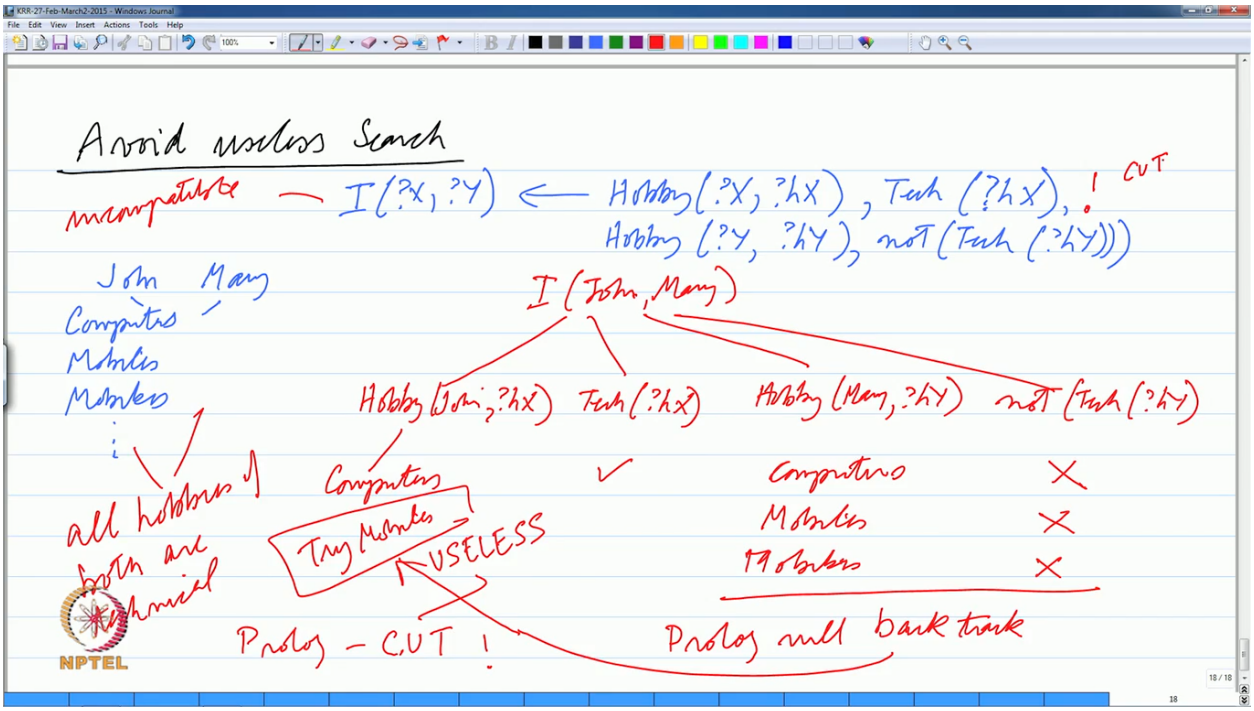
(Refer Slide Time: 19:07)



So we try this. So first we try with computers and find that its technical. Then we go to find hobby of Mary and we try computers and find that it is not technical so I will put a cross to say that it is not technical or in other words not technical is false, because its also technical computer is technical. Then we will try for Mary mobiles and find that its not technical then find mobikes is not technical and if there are only these three hobbies then you have to backtrack. So prolog will backtrack what will it do. It will now go and say try mobiles. But if you now look at this problem a little bit. You can see that when you looked at john's hobby which is computers you looked at all of mary's hobbies and you could not find a non technical hobby. Now what is the point of coming back and trying another hobby of john which is mobiles because again you will find all the hobbies of mary are non technical. So this work is useless and prolog gives you a mechanism for the programmer to signal that don't try this again. How do we do that.

In Prolog there is a statement called Cut which is denoted by an exclamation mark and the so a cut's goal is a goal which is satisfied trivially but which controls the procedural behavior of prolog. So if I were to put a cut here.

(Refer Slide Time: 21:47)



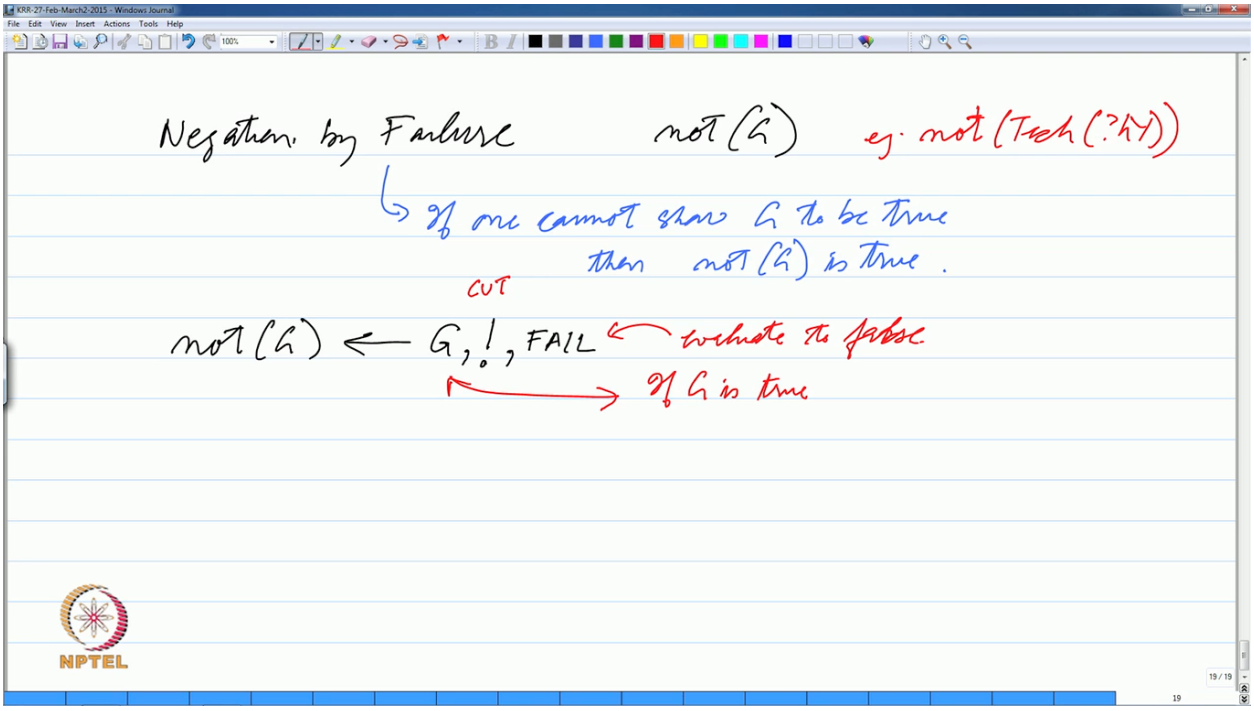
What this is saying is that after you have satisfied the first goal and after you have satisfied the second goal. You have found a hobby for john and you have found it is technical the entire truth value of the incompatibility goal will depend upon the remaining two goals. So in a sense the program can backtrack between the third and the fourth goal you can try one hobby for mary and try to see whether it is technical or non technical, try another hobby but it cannot backtrack across the cuts. So a cut is like a one way gate or its like a what you might say a lakshman-rekha. Once you go from the other side you cannot come back.

And if you have failed to find, satisfy the third and the fourth goal in this case which means you have failed to find a hobby for Mary which is non technical don't try to satisfy the incompatibility goal again. Incompatibility goal is going to fail. So this is the basic idea of cut. And it is useful in many places in making a program efficient.

So let me at this point introduce this negation by failure. Not G so this is in some sense adding to the semantics of prolog and so for example not technical hY we had this goal in the previous example where this is not is what we were talking about. What is this not ? so negation by failure says if one cannot show G to be true then not G is true. Now there is subtle difference between logical negation and negation by failure. Logical negation says that not G is true when you say in logical negation when you say not G is true you are essentially saying that G is false. In negation by failure which is a property of a language like prolog when you say not G is true you are saying that I am not able to show that G is true. So it's a failure to show that G is true.

So we can define negation by failure as follows. So first you try to show that G is true. If you can show that G is true then you can use the cut here so remember that we had talked about a cut. And may be we will talk about a cut a little bit more in the next class. And then Fail. Fail means evaluate to false. So fail is a constant predicate its like false its stating false. So we are saying that not G is true if you can show G to be true. If G is true it means here right. If G is true you will proceed further to the next subgoal.

(Refer Slide Time: 26:16)



Then go to a cut which we have put there and the next thing is return false. In other words whenever G is true this predicate not G will return false. And that is what happened in the John and Mary example we saw. Mary had these three hobbies computers mobiles and mobikes and everytime we tried to see whether they are non technical it returned as failure because each of them was failure. Computer was classified as a technical hobby mobile was classified as a technical hobby, we should just look at that program again.

We had sort of stated that all hobbies of both are technical computers are technical and we were trying to show that we were trying to find a non technical hobby of Mary and every time we tried we found that computers were technical so not G returns false not Tech returns false. Mobiles are technical so not mobile returns false. Mobikes were technical so not mobikes returns false. So if you can show the goal to be true then you return false other wise you return this without any condition so its like a fact which means its true.

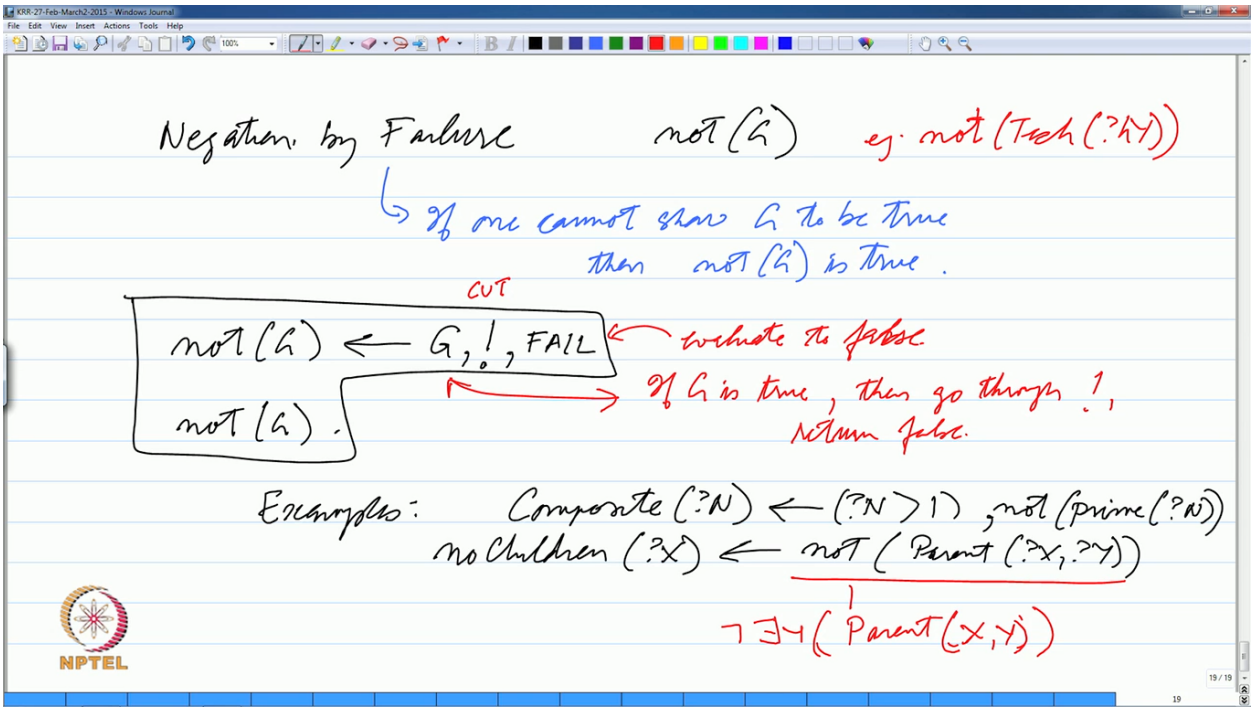
So this is the definition of negation by failure its basically two clauses and notice that we are talking prolog which means order matters. If you had put not G first

then you would always return not G. but now because prolog looks at things in a particular order it will first try the first clause. First clause says try to show that G is true. If it fails to show that G is true then it will backtrack and come to the second clause and say yes not G is true. But if it succeeds in showing G is true it will go through the cut and return false. So not G will return false. So not G will return true only when you fail to show that G is true. So a couple of examples of statements which use the cut. And we will probably look at it again later.

So you can define a number to be composite if the number is greater than 1 and not prime. So if we have a definition for prime numbers already then you can use that definition to define composite number.

You are defining a composite number by saying that a number N is composite if its greater than 1 and its not prime. So that's a definition. And may be we should say greater than 2. But 1 is enough because 2 is a prime so true will not be returned as a composite number, 3 will not be returned as a composite number but 4 will be returned as a composite number. Another example is you can say that X has no children if it is not the case that if X is not a parent. One has to be little careful here this example is given in one of our textbooks which is Reckman and Lewis so you must go and read it up there. But you must be careful that the reading of this is that there doesn't exist Y

(Refer Slide Time: 31:12)



That Y is an existentially quantified variable most of these formulas that we are talking about are universally quantified. Remember that you know our notation here but when the formula occurs on the right hand side inside a not it is treated as an

existential variable. So the reading of this sentence is X has no children if there does not exist a Y such that X is a parent of Y. the reading is not that for all Y X is not a parent. It simply says that there is no Y such that X is a parent of Y one has to be a little bit careful. Okay so we will stop here in the next class we will start by looking at cut a little bit more because it’s a very useful mechanism in prolog. And then we will try to see try to compare forward chaining and backward chaining and try to see whether they are adequate for us or whether they have some problems. We will see that both forward chaining and backward chaining are not complete and we will move towards the resolution method which is a complete method.