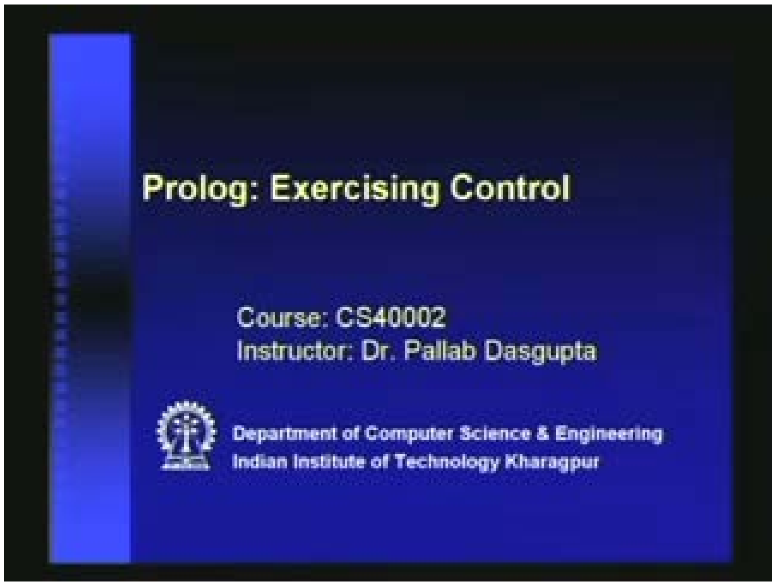
Artificial Intelligence
Prof. P. Dasgupta
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture- 15
Prolog Exercising Control

We will start with our last lecture on prolog. so we I hope to finish off with prolog in this lecture. And in the next lecture, we will cover up some leftover topics from the previous chapters. So I guess that um That will bring us to what we actually want to cover before mid-sems, and then, we will decide that what to do after mid-semesters.

(Refer Slide Time: 00:01:17)



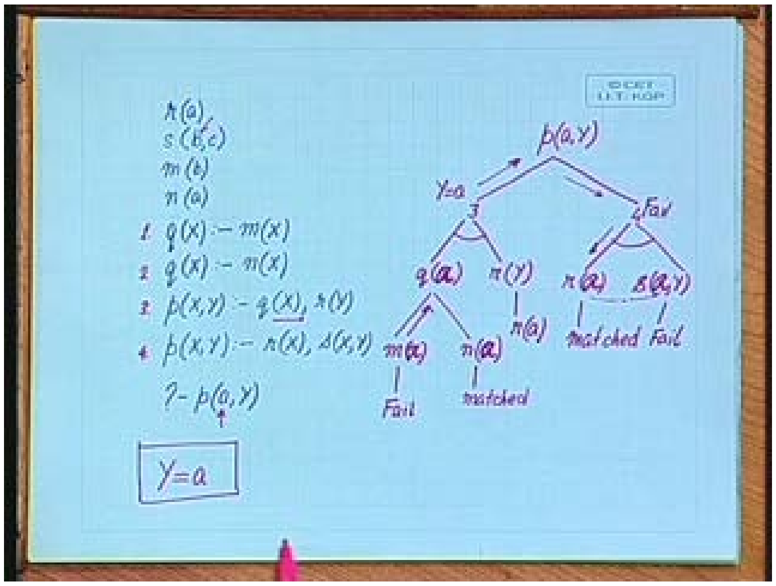
Prolog: Exercising Control

Course: CS40002
Instructor: Dr. Pallab Dasgupta

Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

The chapter that I want to start today is on exercising control over prolog. So far, we have seen that prolog has a well defined way of deducing new clauses from the existing ones. Namely, it does backward chaining and within each rule, it will go left to right among the sub-goals, and do depth first resolving of the goals- left to right- it will carry the bindings from left to right also, on a on each rule. Now, today, what we are going to see is that prolog does allow us some constructs to tamper with this control. We will study those things and that will bring us to the end of what we should know about prolog. I will take one example first, to recap the control flow that we have in prolog.

(Refer Slide Time: 00:09:36)



Let us say, that I have a program which is like this- that I have ra, then s bc-…- these are 4 ground facts, and then we have the following rules: this is just a token example that I am constructing. Now, let us see the control flow that will take place for prolog. We will <interruption in video> And then, there are 2 options here. I can match it with this rule head or this rule head. So, let me create 2 branches. If I take this branch- if I match it with this, my sub-goals will be 2, namely-. Let me number these rules. This is, say, one, 2, 3 and 4.

So, if I choose 3 to instantiate pay, then, I will have 2 sub-goals, namely qx and ry, and I am using the and, because both of these will have to be solved in order to solve this. On the other hand, if I use this rule, then it would have taken 2 sub-goals again, and those sub-goals would be rx and sxy. And whenever we have 2 sub-goals which have a common variable, there is a binding between these variables. So, there is a binding between this x and this x. Now, what prolog is going to do is, it is going to choose the first rule first. It will first take this one- it will first choose this option, and then, within the rule, it will go from left to right, so, it will first choose qx.

Again, for qx, we have 2 alternatives, so, we have another or branch and here, I can have if I choose this one, then it is just mx. If I choose this one, then it is nx. Then, within mx, what do I have? I will have b. But see, you have never told me that I have made a mistake here, all through. See, when I did this pay, and when I got this one, it cannot be qx; it is qa. Then, what about this? This is ra and this is, say, right? When I have this, I will have ma or na. Because it is the instantiation that I started off with- this a, that gets passed on from the rule head to the rule body.

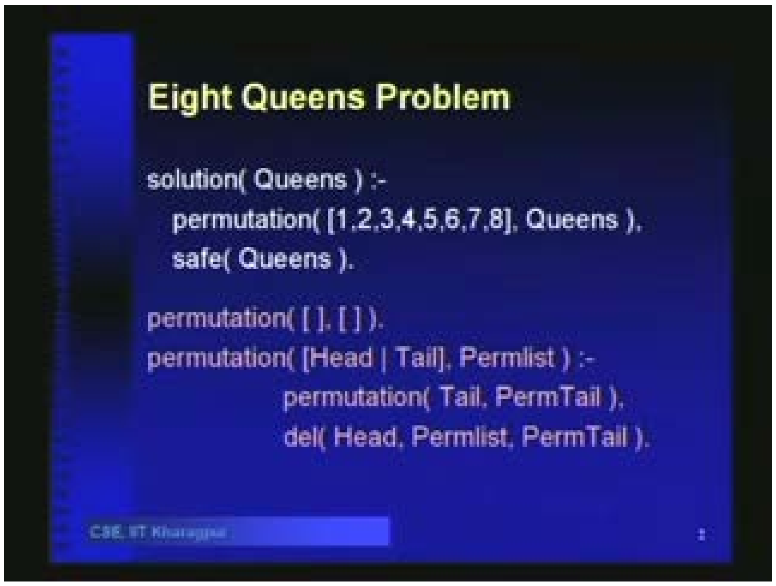
When I come here and then I try to instantiate this with mb, which is the only rule, which has m on the head, then, it does not instantiate- there are no other choices, so fail. So, prolog will fail here. It will backtrack back here. It has another choice here, so it is now going to try na, and indeed, I have na. This is matched na, but we are not done yet, because here, we have an and node, we have yet not solved ry. qx has has been solved, but now, this ry has to be solved. It tries to see what matches- which rule matches r in the head, and there is only one, so we have ra. So, the instantiation that we have here- now, this is solved.

This is solved, so, the instantiation that we have is y, is equal to a, and it gets passed back here. And prolog prints y is equal to a, then, it will try this one, because it will look for more solutions. Then, it will come here and try to solve ra. And ra is there, so matched this has matched ra, then it will go back. It will now try this one, because this is an and node, so unless both are solved, we do not have a complete solution. So, it will go here and try this, and then it will try, s ay, right, but s ay does not match with anything. Here, we have b, so that is not going to instantiate with a, so, therefore, this is a fail.

And when we have fail here, then this and node itself fails. Because if any one of the children of the and node fail, then we are not we have not got a complete solution. We backtrack here; we are back at the root node and we terminate. So, it terminates with only this solution. Now, today, what we are going to study are methods where we will tamper with this flow of the control. We will force it to backtrack from certain points of time. We will force failures at certain points of time. This is in a nutshell- what we are going to see today. So, let me start. Before going to that, we will start with the 8 queens problem.

Let us first see what kind of solution we can have for the 8 queens problem, and then we will go into the control flow. As I was saying, in the 8 queens problem- you remember the problem?- we have to place 8 queens on a chessboard, so that none of the queens attack each other. The predicate which solves this is solution and then a list of numbers- a permutation of the numbers 1 through 8- and what will these numbers give us? These numbers will give us, for each column, the row number where we will place the queen.

(Refer Slide Time: 00:10:07)



Eight Queens Problem

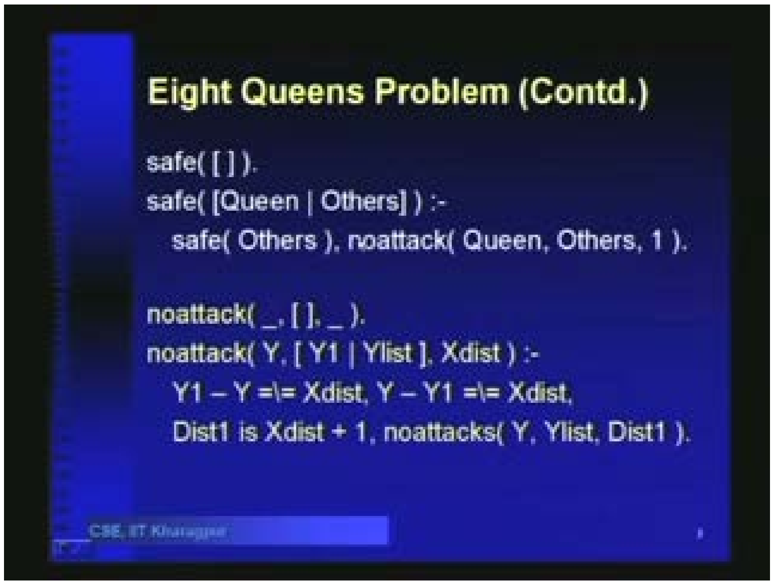
```
solution( Queens ) :-
  permutation( [1,2,3,4,5,6,7,8], Queens ),
  safe( Queens ).

permutation( [ ], [ ] ).
permutation( [Head | Tail], Permlist ) :-
            permutation( Tail, PermTail ),
            del( Head, Permlist, PermTail ).
```

CSE, IIT Kharagpur

The first entry the the This list queens will have 8 queens. The first number in this list will tell us, that which is the row where we are placing the first queen. First queen means the left most queen; the queen in the first column. Second number tells us the row number where the queen of the second column stands, right? That way. So, in order to get a solution, any solution, for this 8 queens problem, will be a permutation. This queens is a permutation of these numbers, that is criterion number 1, which basically encompasses things like, you cannot put 2 in the same column, which is implicit from the formulation. And you cannot put 2 in the same row, because these are all distinct numbers- 1 through 8.

That you cannot have 2 queens in the same column, is implicit from the formulation and you cannot have more than 2 columns in a row and more than 2 queens in a row, is implicit from the fact that it is a permutation of these 8 distinct numbers. But we also need to check the diagonals, so that is why we have this additional predicate- safe queens. This is a permutation of the queens. A queens This queens list is the permutation of this list and it has to be safe. Now, let us see, this permutation I am not repeating, because we have already seen this before. It is the same predicate which is creating a permutation and putting it in queens. Next, we check- what is the predicate safe queens?

(Refer Slide Time: 00:12:40)



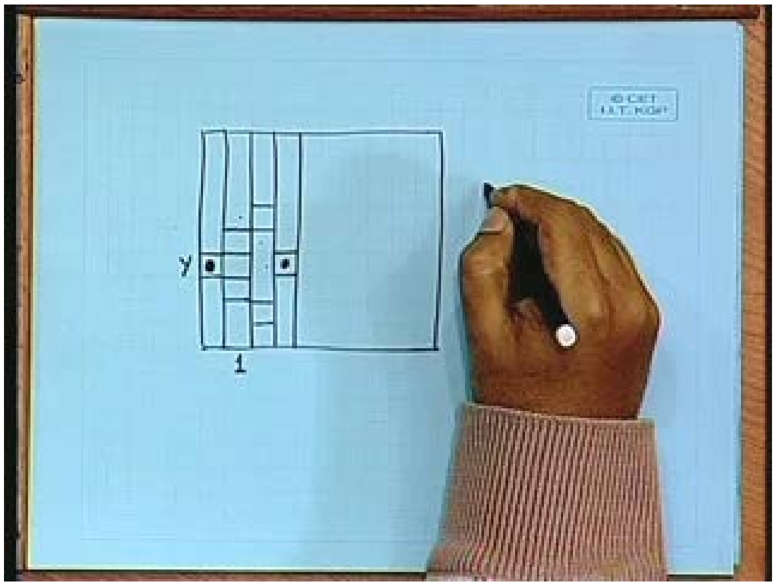
Eight Queens Problem (Contd.)

```
safe( [ ] ).
safe( [Queen | Others] ) :-
   safe( Others ), noattack( Queen, Others, 1 ).

noattack( _, [ ], _ ).
noattack( Y, [ Y1 | Ylist ], Xdist ) :-
   Y1 – Y =\= Xdist, Y – Y1 =\= Xdist,
   Dist1 is Xdist + 1, noattacks( Y, Ylist, Dist1 ).
```

CSE, IIT Kharagpur

The empty lists- for that- we are safe. Then, we are checking safe queen, the head of the list with the others. So, we will This predicate checks that the others are mutually safe. This is the first queen; these are the remaining queens. First predicate checks that the remaining queens are mutually safe within themselves; they do not attack each other. And then, this predicate checks that the first queen does not attack any of the others. This noattack will check that the first queen, which we call queen, is not going to attack the others. And there is this this this curious 1 here, which I will explain shortly. Now, firstly, how do we check that a queen is not attacking the others? This is the policy that we follow. The text text please text please.

(Refer Slide Time: 00:16:33)



What we are going to do is, let us consider that this is the column in which we are looking at the queen. So, this is Suppose we have the queen here in this column. We have the queen here. And these are the other columns, which are to the right of it, and we want to check that this queen does not attack any of the others. So, what we are going to do is, we are going to pick up, one by one, the columns which are at a distance of 1, 2, 3, 4- from this column. Let us look at this column, which is at a distance of 1 from this column. What Which diagonal square do we have to check? If this is in a position of y, then here, we have to check y plus 1 and y minus 1.

When we are looking at the column, which is at a distance of 1 from the first column, and we have to look at distance of y plus 1 and y minus 1, these 2 coordinates- these 2 y coordinates- are important. When we are looking at a distance of 2- when we need to look at y plus 2 and y minus 2, what we are going to do is, in this noattack predicate, we are going to successively check with each of the columns to the right of the first column and for the column at the ith distance, we will check y plus i and y minus i. Coming back to the- yes. (Student speaking).

We are already, I mean- what is your question? (Student speaking). No, see, because we need to check. When you are checking in a particular, say, middle square, let us say, that when we are here- so, your question is, why do not we check these? (Student speaking). Oh, y is already taken care of, because, see, what we have here is a permutation of the numbers 1 through 8. If you have a permutation of the numbers 1 through 8, then you cannot have 2 numbers which are same. So, no 2 columns will have queens on the same row.

That is automatic from the fact that we are permuting the numbers from 1 through 8, to get the row numbers. So, we just need to check the diagonal, right? Let us look at this predicate noattack. So, this noattack This is where we are, so now, it is let us understand what is this 1 here. This 1 is the starting- at the beginning, we want to check with the immediate next column. And then, as we go further down, we will increase this distance and check it with that. Let us look at this second rule first.

So, we want to check whether this y, which is the left most column, that we are checking now, whether it attacks the others, and whether it attacks the column at a distance of xdist. Initially, our xdist will be 1, because we will call with this. This is the distance of the column with which we are checking conflict with the queen in the first column. We checked y1 minus y- is not equal to xdist, and y minus y1 is not equal to xdist. This is 1 is the positive diagonal and 1 is the negative diagonal. And then, we want we also need to check that it does not attack any of the queens to the right of the column y plus xdist.

So, dist one is xdist plus one and recursively check that y- this fellow- does not attack the column at dist 1. Let me repeat this part. In this rule, we are first checking whether the queen, which is at a- which is xdist columns away from the column, where we where we have this y. This is the first column- this is the column xdist away from the first column. So, this is the xdist plus 1th column. Now, we want to first make sure that this queen does not attack the queen that we have in this. So, we check y1 minus y is not xdist. Is this clear? No? What is y1? Yes, y is the queen in the column which we are considering now.

y1 is the column with which we are trying to check conflict, and xdist is the distance of that column from the column where y is in- to the right, always to the right. If it is at a distance of xdist, then we had seen, just now, that diagonally, if you look at that point, then that coordinate- y coordinate- will be y plus xdist or y minus xdist. So, this y1, the queen that is in this column, which is in the y1th row, should not be equal to y plus xdist and should not be equal to y minus xdist. So, that is being checked by these 2 predicates, that y minus y1 minus y is not equal to xdist, y minus y1 is not equal to xdist. But then, we are not yet done, because there are some other queens in this ylist and we still need to check that the queen at this y does not conflict with those queens which are in the ylist.

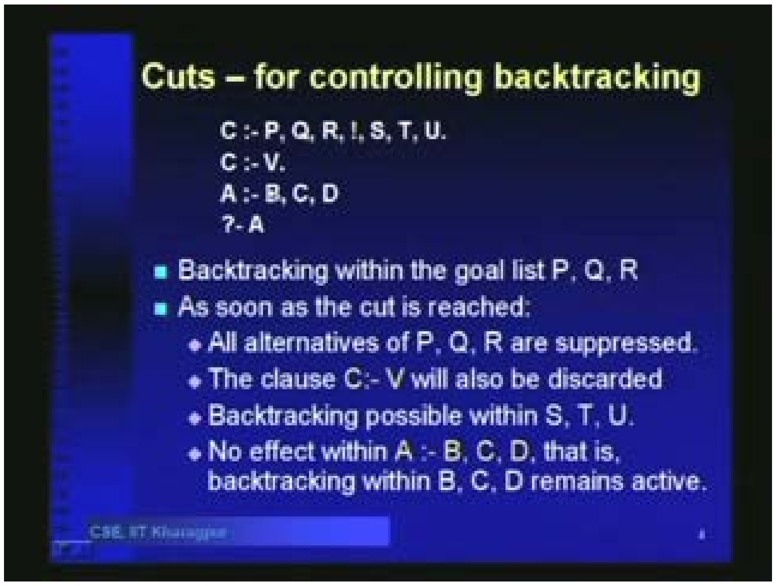
So, we will recursively call noattack on ylist, but the distance will be xdist plus 1, because the next column is that xdist plus 1. dist 1 is xdist plus 1, and then noattacks y this ylist and dist 1- this is recursively going to check that. And how long is this going to continue? Until this y list becomes empty, which means you have already exhausted all the columns, and this is the termination condition, where whenever the middle lists becomes empty, that is a ground clause. So, there is no other attack here.

And we are initially calling it with this distance as 1, so that we can scan all the columns to the right of row. Now, this is, good is it not? Just writing the n queens problem in just 5-10 lines, and it is done. This is a fun of prolog. Now, let us see what kind of control features we use. (Student speaking). No, no, no, not necessary, no. Now, just a minute. Safe- others- (Student speaking). I think it does not matter. (Student speaking).Yes, see, it is going to check- you are not moving the queens any more, within safe. You are not moving the queens any more.

So, it does not matter whether you first check whether the first queen has conflict with the remaining ones, and or whether you check that the remaining ones have a conflict and then check it with the first- it does not matter. Because of the fact that the ones we call safe at that point of time, we are not moving the queens any more. So, the queens have already been instantiated. Let us let us see now. We have things called cuts for controlling backtracking in prolog, and this is a very useful feature, as we shall see shortly.

So, this is prolog program which has a cut here. In this program, I have just written the names of the predicates; I have not shown the arguments, because that is immaterial when defining what this cut means. This cut is this exclamation mark that you see in this rule. Now, what this means is that, once you are you have matched this rule head, then, this becomes your set of sub-goals, and prolog is going to go left to right in these sub-goals. So, it is going to try and first match p. Then it will try to match q, then it will try to match r.

(Refer Slide Time: 00:28:25)



Cuts – for controlling backtracking

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D
?- A
```

- Backtracking within the goal list P, Q, R
- As soon as the cut is reached:
  - All alternatives of P, Q, R are suppressed.
  - The clause C:- V will also be discarded
  - Backtracking possible within S, T, U.
  - No effect within A :- B, C, D, that is, backtracking within B, C, D remains active.

CSE, IIT Kharagpur

Note that these p, q, r, etc., can have bindings. So, there can be several backtrackings between these p, q, r. I mean, solving q is not independent of solving p, because solving p might give you some instantiation to some variable, which is shared with q and q may not have a solution for that particular instantiation. There can be several rounds of backtracking between these- p, q and r, but once p, q and r has been solved, then you encountered this cut, and that means that you cannot backtrack back across these anymore, which means that whatever values that you have instantiated to the arguments of p, q and r.

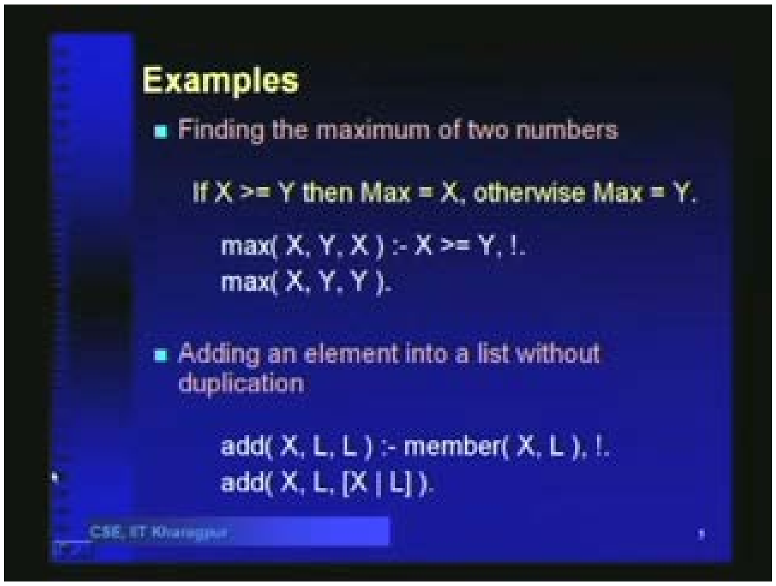
If any of those variables are shared by s, t and u, then those variable values are the ones with which you have to solve s, t and u. You cannot go back and try another instantiation of p, q and r. Now, let us get this thing clear. Let me repeat it. Firstly, do you realize that there can be backtracking between p, q and r? Yes or no? That is because they can have common variables. Similarly, p, q and r can have common variables with s, t or u. Now, once I have solved p, q and r, it means that I have some values for its arguments, for which p, q and r is known.

Then, those arguments values will also be bound to s, t and u, provided s, t and u has any of those variables. Now, because of this cut, what this cut means is that once you have passed this cut, if, for that particular values for s, t and u, you do not find a solution, you cannot backtrack and try another value of pqr. Which means, if you do not find a solution for s, t and u with those values, then this whole rule will fail, and not only will that rule fail, this other rule- c to v will also fail. So, you will be back here, and you have to try a different instantiation of b. Should I- is this clear?

Okay, so let us see what we have. We can backtrack within the goal list pqr- no problem. We can try out different instantiations, different ways of solving pqr. As soon as the cut is reached, all alternatives of p, q and r are suppressed. The clause c to v will also be discarded. Backtracking is possible within s, t and u, but not across the cut. So, within s, t and u, we can try backtracking and trying on various things, but you can never go back to p, q and r for a different instantiation. That is not possible and also there is no effect within bcd.

If with that instantiation, you are not able to solve c, then you come back to b and then try a different instantiation, different solution, for b, and then again go into c. (Student speaking). With s, t and u, or with s- (Student speaking). No, when you go back to b, at that point of time, there can be some other- suppose b does not share any variable with c. If b does not share any variable with c, then you fail all together, right? But if there is a variable which b and c shares, then, maybe, that b can give you a different instantiation of that variable, for which c will be able to solve p, q and r with another instantiation. So, that is tha That is a possibility which will necessitate you to check whether there are any other alternative solutions of b. Let us look at some examples of cut.

(Refer Slide Time: 00:32:30)



Examples

- Finding the maximum of two numbers

  If X >= Y then Max = X, otherwise Max = Y.

```
max( X, Y, X ) :- X >= Y, !.
max( X, Y, Y ).
```

- Adding an element into a list without duplication

```
add( X, L, L ) :- member( X, L ), !.
add( X, L, [X | L] ).
```

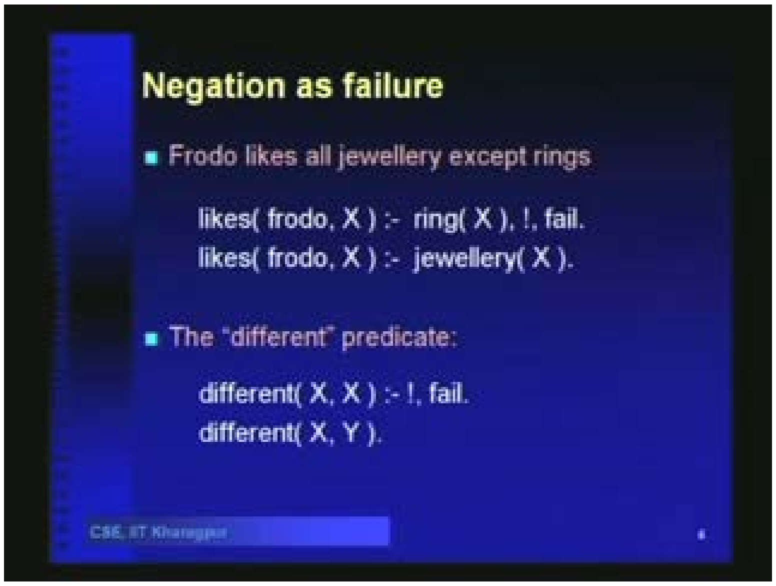
CSE, IIT Kharagpur

Suppose we want to find the maximum of 2 numbers. What we want to check is, if x is greater than equal to y, then max equal to x, otherwise, max equal to y. We will write it in this way. The predicate max xyz is going to instantiate z with the maximum of x and y. max xyz is going to instantiate the maximum of x and y with z. So, if x is greater than equal to y, then x, right? So, max x,y,x, if and if x is greater than equal to y, and then cut. Because if you already have this, then you are not going to try for any other solution, right? And then otherwise, the maximum of x and y is y.

Prolog is going to try this first, and then this. So, if this works, then it is going to cut it there and not try this. If that does not work, then it will try this, and this is always going to work. Is it clear? Now, what would have happened if I did not use this cut? How should I How would I have written? (Student speaking). We have to put again: y is greater than equal to x, then it will match this one once. If they are equal, it will match this one once, will match the second one once and print the same thing twice.

So, this is a neat way of writing that another example; adding an element into a list without duplication. This is how we will write. So, if x is a member of l, then cut- this means you discard, right? If it Otherwise, just add it to the head. There is an even more interesting application for cuts. Let me see. Yes, for cuts, which is called negation as failure.

(Refer Slide Time: 00:39:06)



Negation as failure

- Frodo likes all jewellery except rings

```
likes( frodo, X ) :- ring( X ), !, fail.
likes( frodo, X ) :- jewellery( X ).
```

- The "different" predicate:

```
different( X, X ) :- !, fail.
different( X, Y ).
```

CSE, IIT Kharagpur

See, so far, in prolog, we have not talked about the not operator. We have and in the sense that sub-goals are anded we have or, because we have or between the rules. But we have not talked about not and indeed negation is a very tricky thing to handle in logic programming. Or, and for that matter negation, is the source of the complexity in all these logics. Let us look at how we handle negation in prolog. Suppose we want to express that Frodo likes all jewelry except rings. We know what happened with Frodo and the ring, so it is quite natural, right? Now, this is 2 rules. The first one says, okay, now we have a predicate here which is called fail. See, we have this predicate here called fail. What does this fail do?

This fail forces the program to fail at this point. So, whens When you go past the cut, then this is going to be taken as failure and not success. This is unlike the previous case, where after the cut, we had the full stop, which means that after the cut, it was saying that done solved it was saying that this is solved, but here, it is not saying it is solved. Here it is saying that if you go past this cut, then that is a failure. So, this is actually expressing the opposite thing. We are first checking out whether x is a ring. If x is a ring, then Frodo does not like ring. If you ask likes Frodo ring or likes Frodo x, where x is a ring, then it is going to first check these, and immediately say fail. It does not like it, right? Otherwise, if x is a jewelry, then Frodo likes it. Is it clear, the use of failure?

We are actually using the negation with this fails. Whatever we want to check, that this should not happen, we will write that and put a fail at the end of it, so that whenever that matches, you will directly return failure. But note that this fail is only failing for this rule. Why did we use this cut? We have to understand that. Why did we use this cut? If we did not use the cut, it would mean that when it is a ring, it will fail, and fail is going to force you to backtrack from there. But it will again go and try this one, because this is an alternative. So, it will go and try this one, but this cut is preventing that you have gone pass the cut and then failed.

Therefore, that is total failure. That is that is total failure for this predicate likes, including the alternative predicate here. For example, let me go back to the- (Student speaking). If you want to express a set which is negative, then you write that program, check for the negation and then put a failure at the end of it. Now, if you want that if this failure occurs, then that whole predicate should be discarded and we should start back from the previous predicate, then, you have to put a cut there. If you do not put a cut, that particular rule will fail and you will try other rules, which has the same rule head.

For example, in this program that we were seeing earlier- slides please. Yes, in this place, if we had put a fail here after the cut, that would have meant that whenever you go past this cut, whenever you are able to satisfy p, q and r, then, you will fail and that failure- because of the existence of this cut- it is going to take you directly back to b. That would mean that find out an instantiation of b, which does not satisfy p, q and r, but satisfies v. That would be what is meant by putting a fail after this cut. Just think it over a little more, then you will get a feeling that what is happening.

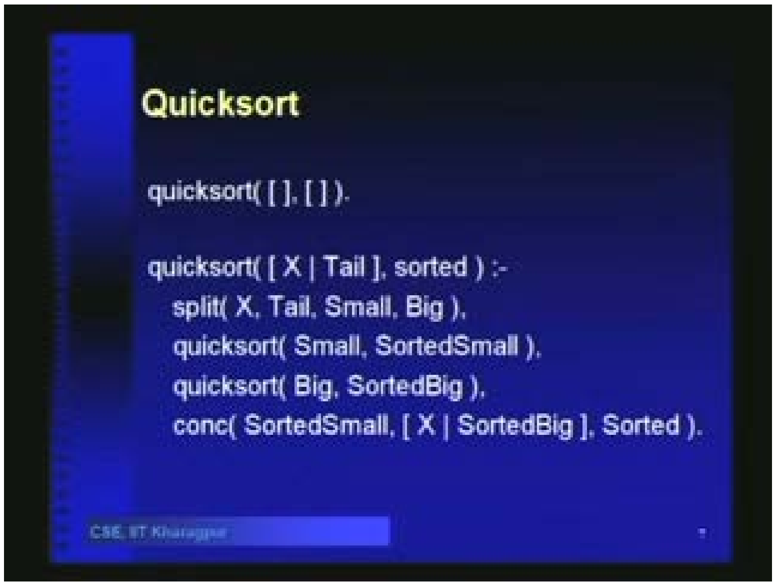
Fail predicate is just going to make it backtrack along with the cut. It is going to make you fail the whole predicate. Yes. (Student speaking). Yes, the second one will be tried when you have already exhausted all possible ways of solving the first row. And that is what we will want, if we put the fail after the cut. And then, we have this very useful predicate- the different predicate, which checks whether 2 values are the same or not. And we can write it like this. See, this is one way of checking. This is the implementation of x not equal to y.

Recall that in predi in prolog, x and y can be anything. It can be variables, lists, compound lists, whatever. This is a clean way of checking whether they are different, so if you are able to match the first rule, namely different xx, then you fail and fail after the cut, so that is the end of it. They are not different, so it will return failure, otherwise they are different. (Student speaking). Even- (Student speaking). Yes. No, no. Here, you are first checking out whether it is a ring. See, prolog is going to try out the rules in this order. So, if you like If you put likes Frodo abc, where abc is a ring, it is going to first test out whether ring abc and ring abc will succeed, so it will fail all together and say no.

If you ask likes Frodo abc, it will say no, because ring abc has matched. (Student speaking). No, because you have this cut. You are not going to try out anything else. That is why we have the cut and the fail. Cut will prevent you from trying out the alternatives and fail is going to fail you here. So, you cannot try out other alternatives because of the cut and you have failed here, so it is total failure, right? And in the second case, if this ring x fails, if there is no instantiation of ring abc, if ring abc is not there in the knowledge base, then, this rule will not succeed.

I mean, you will not get any successful instantiations of this. So, you are going to try the next rule and the next rule is going to check whether jewelry x. We know it is not a ring and if it is jewelry, then it is a jewelry other than a ring, and then, Frodo likes it. Is that clear? Now, only thing that you have to be cautious about here is that you must remember that this cut is going to only prevent you from trying out any of the rules which has likes as the rule head, which has the rule head, right? But it does not prevent you from backtracking across predicates at some other level in the decision tree, in the proof tree. You remember in that earlier example, we had that abc, and then c had these 2 rules, but the backtracking within abc is not prevented by the cut? Even you have cut and fail, it is going to fail the c part, but that backtracking within abc will still be the same. Quicksort.

(Refer Slide Time: 00:42:50)



Quicksort

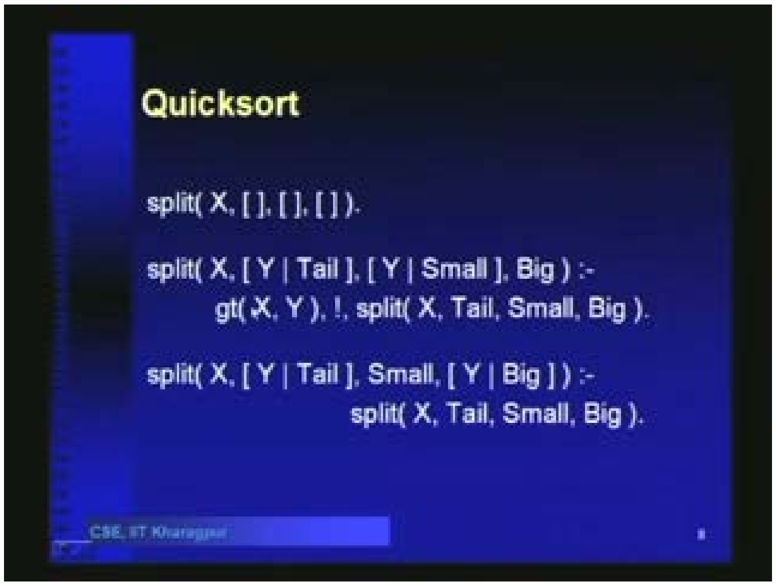
```
quicksort( [ ], [ ] ).

quicksort( [ X | Tail ], sorted ) :-
   split( X, Tail, Small, Big ),
   quicksort( Small, SortedSmall ),
   quicksort( Big, SortedBig ),
   conc( SortedSmall, [ X | SortedBig ], Sorted ).
```

CSE, IIT Kharagpur

I think you just- this part is easy. This part actually checks that quicksort x, followed by tail sorted. It will fi First step is splitting it. It is going to split this thing into small and big. Then, we will quicksort small to get the sortedsmall list, quicksort big to get the sortedbig list, small and big are the left and the right. Small is the left part of the array around the pivot, and the big is the right side of the array around the pivot. And then, we will concatenate the sortedsmall with sortedbig, but x here right x is the pivot.

So, sortedsmall and then x followed, sorted by sortedbig to get sorted. This part- this last 3 parts is simple. This is the predicate, which does all the work. This is the splitting. Concatenate, we understood. We have seen previously, and this is just recursively calling it on the left su left um part of the array and the right part of the array. This is the main thing. Now, just take this down. This is the split thing. This is just jugglery with the list to get your elements into small and big respectively.

(Refer Slide Time: 00:44:35)



Quicksort

```
split( X, [ ], [ ], [ ] ).

split( X, [ Y | Tail ], [ Y | Small ], Big ) :-
        gt( X, Y ), !, split( X, Tail, Small, Big ).

split( X, [ Y | Tail ], Small, [ Y | Big ] ) :-
                    split( X, Tail, Small, Big ).
```

CSE, IIT Kharagpur

We have this cut here, because if you find that x is greater than y, then you do not need to do the other rule, you do not need to check the other rule. If x is greater than y, then you just need to split this part. Otherwise, you need to- here, it is going into the left; here, it is going into the right. So, if x is greater than y, then y should go into small. So, it is going into the left list. if if Otherwise, y will go into the big. That is all. It is just taking each element and putting it either in small or in big. So, first one is putting it in small, and the second one is putting it in big, and recursively, you do the split of the remaining element.

Now, is this clear what is happening? (Student speaking). Ya, see, I have split x, then this is the array which I am splitting, and this is the small and this is the big. So, first argument is the element, the pivot element. Second argument is the original list. Third argument is the left half left half or whatever left part, and the fourth argument is the right part. I am checking x with the first element of the original array, and if x is greater than that element, then this element should go into the left one. See, we have y in the small, and in that case, we put a cut and then just recursively split the remaining part of the original array.

What do we mean by split the remaining part of the original array? The remaining elements in the original array will also have to be distributed into small and big. So, the remaining part is this tail and the remaining part of the small lists is the small; remaining part of the big list is this big. And for the other one, if x is not greater than y or y is greater than x, whatever, then we will try this rule. If this succeeds, then the other rule is discarded, because of the cut. But if this does not succeed, then we go into the other rule, where we put the y element in big.

Small remains as it is; y goes to big. And then recursively, we split this tail as small and big. Previous slide- (Student speaking). No. It will not- see, this is this split x tails small big predicate will succeed only if all elements of tail which are smaller than x are in small, and all elements which are bigger than x are in big. See, the thing that- in prolog, computation and instantiation are the same thing. So, if you give a instantiated stuff, it is just going to check its correctness. If it is not given, then it is going to try to find an instantiation which satisfies that. So, all we are doing is, we are declaring that when is split predicate successful.

Split predicate is successful when the first argument x and tail, and small and big are such, that all elements of tail which are smaller than x, should belong to small; all elements of x that are greater than or equal to x should belong to big. And split is doing that with these 3 predicates. The first predicate says, that if the first element of tail is greater than x, rather, less than x; if x is greater then the first element of tail, and the first element of tail is also the first element of small, which it should be, and if we recursively apply split on the remaining thing, then we will get the small and big, then this split is correct; when this split is satisfied.

Similarly, for the second one- (Student speaking). This is satisfied, and these sub-goals are giving you an indication on when this split is correct. So, if you see how this is going to actually work, when small and big are not instantiated, is that it is going to go down recursively breaking this down, right up to the bottom. And then instantiations will start coming back from there, because of this split x empty. (Student speaking). In quicksort? The second argument is the sorted list. First argument is the original list, second argument is the sorted list.

Slides please. First argument is the original list and second is the sorted list. So, if you give the original list and sorted list, it is going to say true. If you give the original list and an unsorted list, it is going to give you false; and if you give the original list and a variable, then it is going to return in the sorted list. So, with that, with that we come to the end of our discussion on prolog, and I hope you enjoyed it.