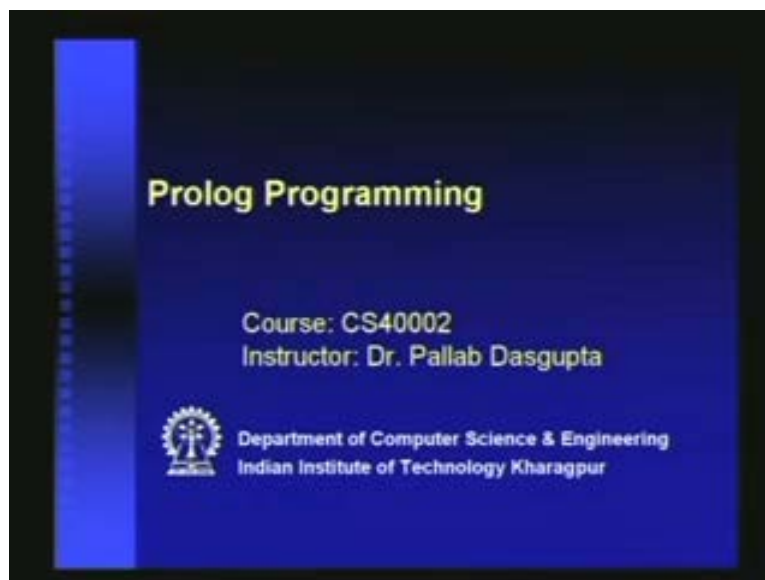


Artificial Intelligence
Prof. P. Dasgupta
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture- 14
Prolog Programming

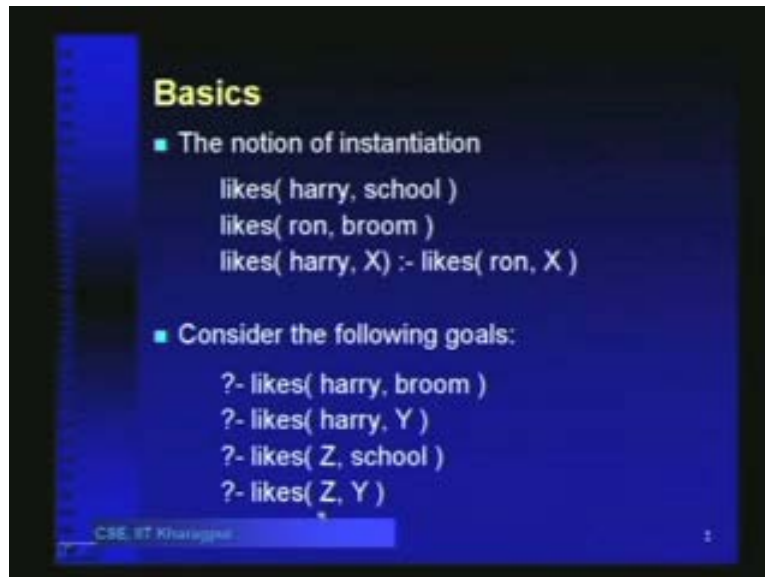
so please Please get seated from this side, yes.

(Refer Slide Time: 00:00:53)



We will continue our study of the prolog programming language. In the last class, we had introduced the basics of the language. To give you a brief recap, we can have in prolog, facts, wrong facts specified like this, the predicates like this, and also, we can have rules like this- likes Harry x if likes Ron x, and then, we can deduce goals like likes Harry broom, which is a ground goal, ground predicate, and we just have to find out whether this is correct or not. So, we have to find out whether likes Harry broom can be inferred from the set of facts and the given set of rules.

(Refer Slide Time: 00:02:06)



The slide is titled "Basics" and contains the following content:

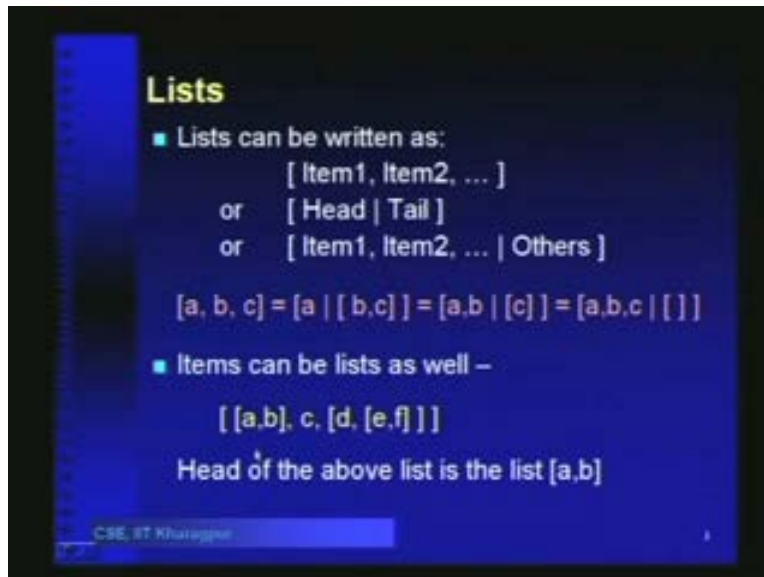
- The notion of instantiation
 - likes(harry, school)
 - likes(ron, broom)
 - likes(harry, X) :- likes(ron, X)
- Consider the following goals:
 - ?- likes(harry, broom)
 - ?- likes(harry, Y)
 - ?- likes(Z, school)
 - ?- likes(Z, Y)

At the bottom of the slide, there is a footer that reads "CSE, IIT Kharagpur" and a small number "1" in the bottom right corner.

We can also give queries which involve a variable, in which case, prolog will find out all instantiations of the variable which satisfy the predicate. For example, here, it will find out all instantiations of y which satisfies likes Harry y . For example, y will be instantiated to broom as well as school, in this program. And then, we had also seen that you can actually give a goal where all the arguments are variables, in which case, it will find out all ways of satisfying that predicate. Let us recap what we had studied on lists. We had seen that lists can be written in several ways in prolog, and lists are, in fact, the only kind of data structure, **if mi** if I may call it that, that prolog supports.

So, the lists are of the following types: you can have item 1, item 2, etc., or you can have head followed by tail; you can have item 1, item 2, and then list, so whatever comes after this bar is another list, which is the rest of the original list. And then, we had seen that items in the list can, in turn, be lists themselves, right? After this, what we are going to do is, we are going to study several examples of prolog programs of which we had started seeing 2 programs, namely membership in a list, which was like this, that this member-predicate has 2 arguments, x and y , and is true if x is a member of y .

(Refer Slide Time: 00:03:08)



Lists

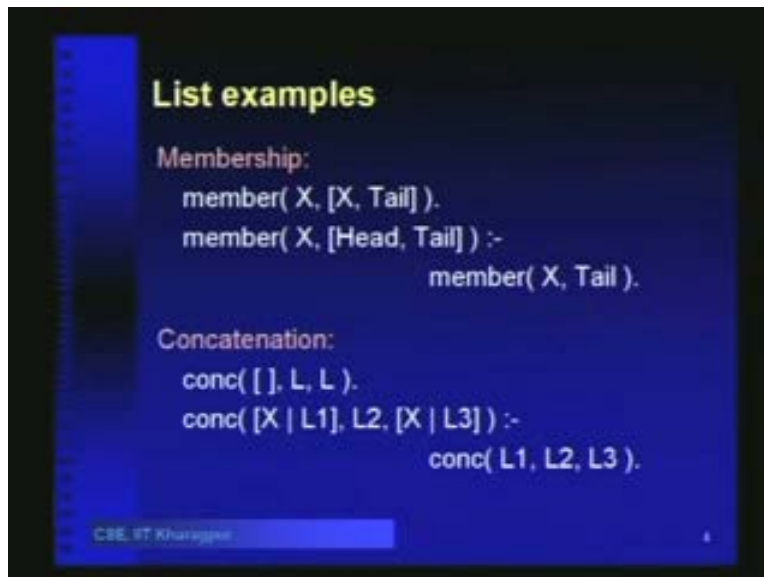
- Lists can be written as:
[Item1, Item2, ...]
or [Head | Tail]
or [Item1, Item2, ... | Others]

$[a, b, c] = [a | [b, c]] = [a, b | [c]] = [a, b, c | []]$

- Items can be lists as well –
[[a, b], c, [d, [e, f]]]
Head of the above list is the list [a, b]

CSE, IIT Kharagpur

(Refer Slide Time: 00:05:02)



List examples

Membership:
member(X, [X, Tail]).
member(X, [Head, Tail]) :-
member(X, Tail).

Concatenation:
conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
conc(L1, L2, L3).

CSE, IIT Kharagpur

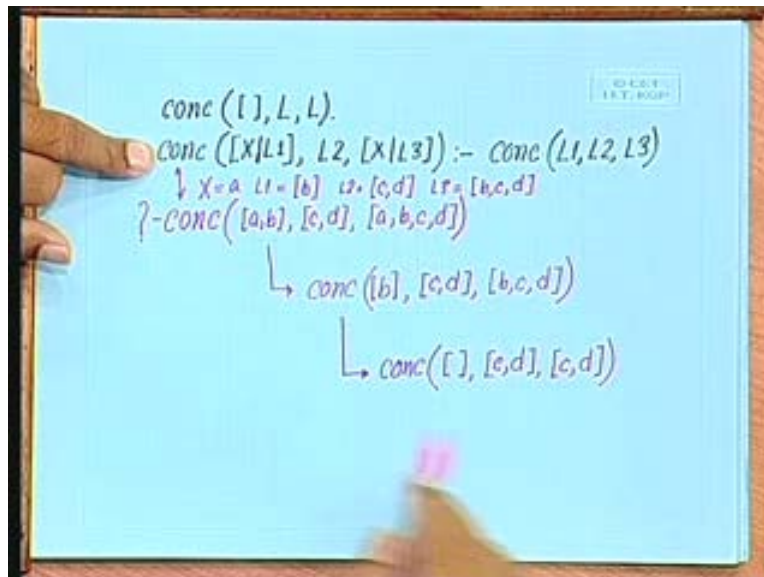
And when we refer to member of y, we just mean that at the top level list. So, we assume that in the predicate member xy, y is a simple list, or we are actually finding whether x is an item of the list itself and not any sub-list. So, under this, we had 2 predicates- one is member x, x comma tail, which is satisfied when the first element of the list is x. And

then, we have member x comma head comma tail if and only if member x tail. So, if the first predicate fails, that means that the head does not match with x, and therefore, we find recursively find x in the tail of the list. It is a member only if it is a member of the tail. Then, we had seen, concatenation- did I do this in the last class? Concatenation?

So, to quickly recap- concatenation is in 2 predicates: the first one says that if we concatenate an empty list with l, then we get l, and the second rule says that if we concatenate x as the head and L1 as the tail with another list L2, then, we will get a list x with L3 where this x is bound to this x. So, the first element of this list is the first element of this list and the concatenation of L1 one and L2 will give us L3. Yes. (Student speaking). Brackets. (Student speaking). No, L2 is a list which is already a list. See, this this concatenate predicate has 3 arguments, all 3 are lists, is that is that clear? No? Give an example of how this will work? Okay, let me first take down the predicate. Concatenate this L1, then L2.

Now, let us first take an example where we are looking for, say, the truth of the predicate. So, we are given 3 lists- so, we are given, let us say, ab and we are given cd, and we have abcd. Suppose this is our goal. What is prolog going to do? It is going to try to match this with the rule heads. So, the first rule head is this- this is the ground fact. Does this match with this? Is there a unification between this and this? The answer is no, because this empty list cannot unify with this list, which is non-empty. This unification will not work, so it is going to try this one. So, if we unify this with this, then, here, x will be equal to a, b will be the list.

(Refer Slide Time: 00:09:54)

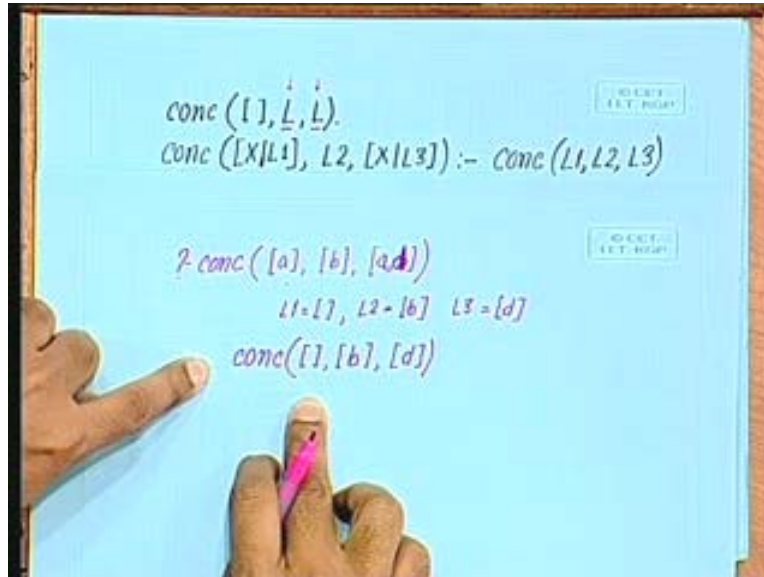


If we unify this with this, then, x will be a- just the element a, L1 will be the list b, L2 will be the list cd, and then x will unify with a, right, and L3 will be the list bcd. So, once this head matches the sub-goal, that will spurn off from this, will be this one conc with L1, which is b, right, b. This b, yes, L2, which is right, and L3, which is- right? Now, this is our sub-goal. With this sub-goal, we will again try to do the same thing. Again, it will not unify with the first rule head, because this is still not an empty list. Again, it will unify with the second one, and that is going to spurn off another sub-goal and what is the new sub-goal going to be?

It will be conc, empty, then cd and cd. Then, it will try to match this again with the 2 rule heads. This time, the first rule head will match, because this empty list unifies with this empty list, and these 2- this 1 can unify with these 2, because these 2 are identical. (Student speaking). When they are equal, it can identify. Therefore- (Student speaking). It will do all that. See, the point is that, here is a binding- the binding between the second and the third argument of this predicate. So, it says that whatever you instantiate the second argument with, will also have to match with the third argument for this rule head

to unify with whatever we want to unify, with this. For example, if we slightly modify this case and look for matching.

(Refer Slide Time: 00:13:30)



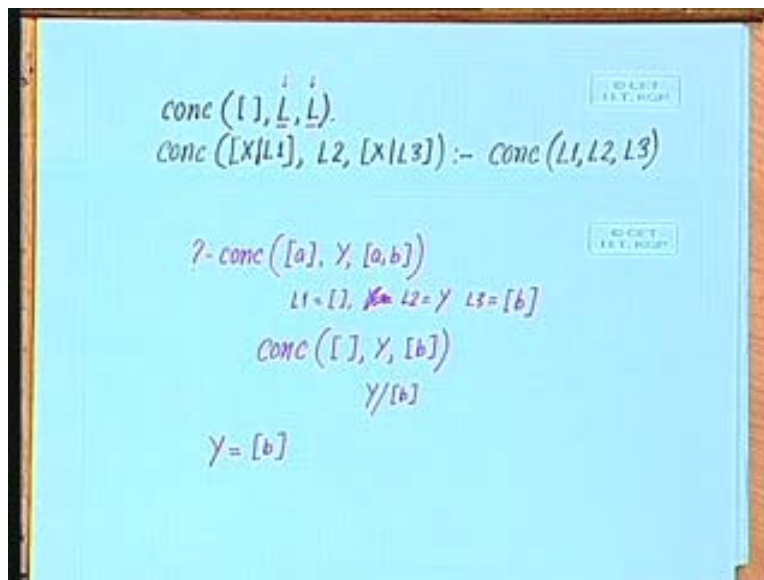
Suppose we try to check whether conc, say a and b, will work on that. Again, if we try to do this, we will first try the first rule. It will not match, because of these. **it will all it is it is not** It will not match because of these, and also because of this, because these 2 will not unify. Therefore, it is going to try the second one and when it tries the second one, again, this binding cannot be satisfied. The head of the first argument and the head of the third argument are not identical. Therefore, this rule head will also not match and when both of the rule heads do not match, then, you will return false, because there is no other option to unify this one.

If you had something like, say, ab, with a, say d, let us see what will happen here. It will not unify with this, so, it is going to try with this one, and then when it tries with this one, now this binding is there, right? So, it can unify with this, and the unification will give- it is going to give L1 is empty list, L2 is the list b, and L3 is the list d. The sub-goal that we are going to have is... so, once it matches with the rule head, this becomes a set of sub-

goals. If we can prove these, then we are done. If we try to prove this, then what is going to happen? We will try to prove empty, L2 was b, and L3 was d. We go back and start again.

This time, the first argument unifies, but these 2 do not unify, because if I unify one with l, then I cannot unify the other with l. So, this binding cannot be satisfied by this predicate. Therefore, it does not match with this rule head. We go to the second one, and in the second one, the head of this one is empty, whereas the head of this one is d. Therefore, again, this binding from x to x here cannot be satisfied here. Therefore, it does not match with the second rule head either, and that is when we declare that it is a failure. Is that clear? Just to round it up, let us look at one more case, where we have a free variable.

(Refer Slide Time: 00:16:40)



Suppose we say, that what is going to happen if we have conc of a with y and we have ab? Let us see what is going to happen. It is first going to try this one; it will not unify, because of the first argument. It will try this one. Now, let us see how the unification proceeds. It is going to try to unify the head of this list with this x here. Head of this is a,

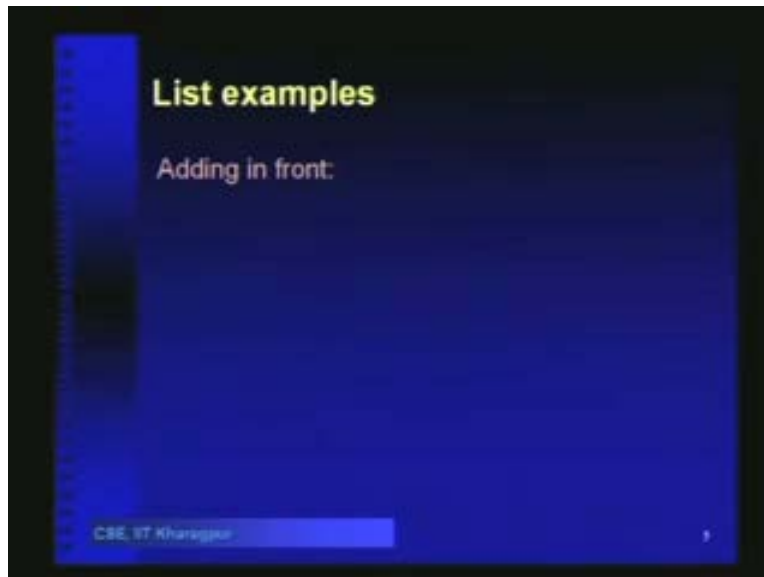
and then it will check out the binding with the third argument, and yes, the binding can be satisfied, because this head is also a. Therefore, it is going to create a unification, which is as follows: L1 is the empty list, y is- rather, L2 is- the free variable y, and L3 is b- the list b.

Now, once we have this, we will try the sub-goal empty, y, b, and again, it will try with the first one. This can unify with this, right? Now, this can unify with this, and we can do the unification, if we unify y with the list b. So, **if we rep** if we substitute y with the list b, then these 2 will unify. Therefore, prolog will print- (Student speaking)- in which case? In the first term? (Student speaking). Yes. (Student speaking). No, it- in the first round or on the second round. (Student speaking). In the second round, it is unifying with the first one only. We are unifying with this only. (Student speaking). Yes, it is going to give me an instantiation of y, which is able to satisfy this predicate. (Student speaking). Oh, after this? Yes, yes. It will.

It is not done yet, **after fi** after printing this, it will still try to match this with the second one, but the moment it tries to match it with the second one, this empty list cannot unify with this, because here, we have x: here, it is empty. (Student speaking). Yes, but then your L1- (Student speaking)- no, this is not going to unify with this one, because both are empty. Then, both has to be empty. This unifies that at- I mean, x cannot be an empty element, if you write it this way. x is not a list, x is an item. And when you are talking about empty, it is the empty list. So, x cannot unify with an empty, because they are 2 different types- empty is a list type and x is an item.

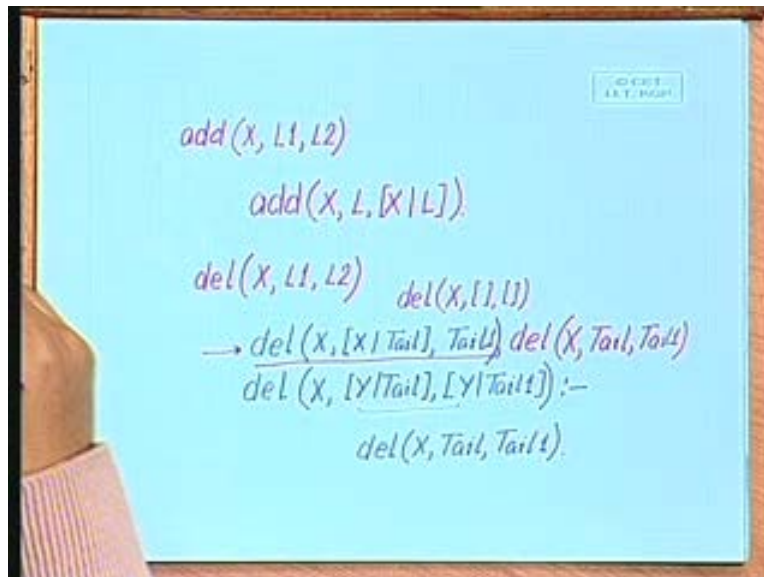
Is this clear? The way in which prolog is working backwards? (Student speaking). Okay, it does not **it does not** distinguish between- there is no explicit declaration as such. But when you are specifying that x bar L1, like here, that the semantics of the language tells us that this is the list and this is the head of the an list, so it is an item, okay? Let us do a few more examples and try to work out a few yourself.

(Refer Slide Time: 00:19:48)



So, let us see. Suppose we are given a list and we want to add an element to the front of the list. **so ok so we have** Text please. **text please text please text yes**. We have add an element x into a list l to get another list. This thing- how are we going to write this? Add x to $L1$ to get $L2$. That is the same in prolog, as saying that the concatenation of x and $L1$ is $L2$, except that here, unlike the list, concatenation here- x is just an item and we want to have a list $L2$, where x is the head and $L2$ is the tail. How are we going to write this? Try writing it. Firstly, write the ground part of it, **so the** the termination condition for the recursion.

(Refer Slide Time: 00:27:57)



Add x to $L1$, is it not it anything else is to be done? That is it. that is it Nothing else needs to be done. What about deletion? Suppose we have to write deletion of x from $L1$ gives $L2$; deletion of x from $L1$ gives $L2$. How are we going to write it? (Student speaking). No, no, no. Here, x is not necessarily the front element. It is not necessarily the head, but in some element of $L1$ and we want to get $L2$ after deleting. If $L1$ does not contain x , then $L1$ and $L2$ will become identical. Again, the termination condition will be worth when we find, so that is the termination condition. Let us first write down the termination condition. So, the predicate for the termination will be- delete x , x comma tail and tail.

If x is the first element of the list, then after deletion, we get tail. Otherwise, what should we do? Otherwise, we have to look further into the list. We just have to skip the head and look for the deletion in the tail. We will write delete x and then y tail, and if the first element is not x , then the first element of $L1$ and the first element of $L2$ has to be the same. Therefore, we have to put y again here, so this is the binding, followed by some tail 1. So, $L2$ some other list, right? So, this is- if delete- right, [noise] so we are now beginning to get a hang of how prolog works, right? (Student speaking). If $L1$ contains-

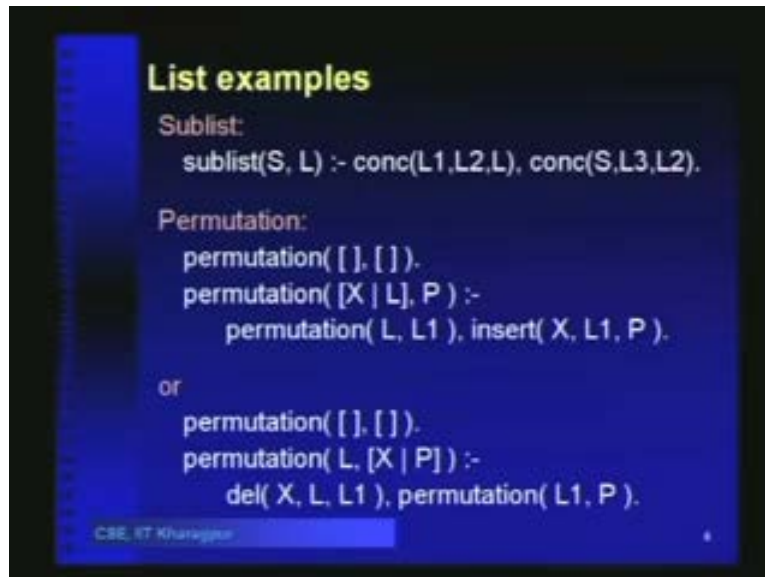
(Student speaking). In this case, it is going to remove only 1 of that. (Student speaking). Yes, it is going to delete the left most one.

But you can always write another predicate that will delete all instances of x, so that is a nice exercise. Try to extend the delete predicate, so that it deletes all instances of x from the list L1, to return the list L2. (Student speaking). So, you have to continue up till you- yes, until you get, actually, so this empty list check will come before this, and here, after checking this- the slides please, the text. **text text sorry yes** After getting this delete, this thing, right, **so this is this will continue**, so, this is not going to be the termination criterion anymore.

Once you get this this thing, you will also have to add here that delete; with this, you have to then add delete x with tail. (Student speaking). No. That will be- **the the the** the termination criterion will be that when we have- delete x, and we have the empty list here, and the empty list here. (Student speaking). Now, what if the last element is not x? You have to go to a point where you are deleting x from an empty list, and the result is also an empty list; that is going to be the termination criteria. When you have are here, that you have found one instance of x, then you still have to delete.

I mean, this is not yet tail. It is some tail 1 still, because it is not yet deleted all the elements from x. This is going to be some tail 1 and you will have to **do** delete tail L1. This is another rule that is going to come up, so try to write it out- deletion of all instances of x from L1 to get L2. Now, let us look at another case: sub-list. The sub-list- case of l, concatenate L1, L2 to get l, and concatenate s, L3. Can you follow what is this happening? We want to find out whether s is a sub list of l.

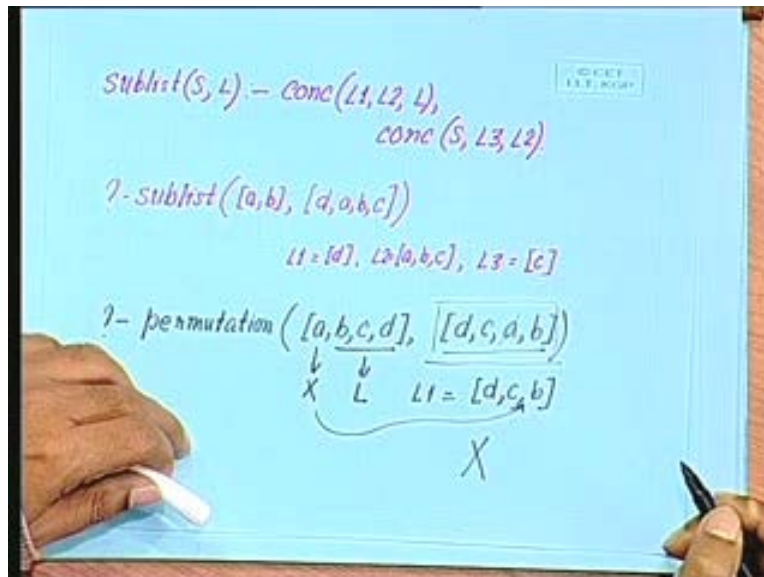
(Refer Slide Time: 00:42:48)



So, that is going to happen, if I can find out 2 lists- L1 and L2, such that their concatenation gives l and concatenation of s with some L3 gives us L2. (Student speaking). Now see, why is this so complex? **The com the** It is so complex, because the lists elements can in, in turn, be lists. We are just trying to find out whether s is a sub-list of l, so it could be- within l, it could be inside some level of nesting. So, what we are trying to do here is, we are trying to see whether it is a sub-list, then, there is some s.

If we concatenate s with L3, we will get L2 and L2 and L1 will give us l. So, at one level of nesting- (Student speaking). So, let me execute this with 1 example. Let me first take this down. Sub-list sl, if concatenate L1 L2 l and concatenate s L3. Now, this concatenate predicate is already defined previously, so I am not repeating that. Let us say, that we want to find out whether- so, here, when I try to match with this, then, what is happening? So, L1 is d, L2 is abc, and then, when we come here, try to unify with this. Then, what is L3?

(Refer Slide Time: 00:42:20)



Roughly, this is the way in which the final instantiations are going to take, but the question is, how does it know that it is going to break up in this way? It does not know.

What it is going to do? Is it- will try all kinds of instantiations progressively, until it hits upon the right instantiation, and is able to match the goal. Let us look at permutation. Our objective is to check whether one list is the permutation of another. So, permutation of the empty list is the empty list. And then, observe this thing. Now, let us see- what is the second rule telling us? Second rule tells us that- so, the first argument is the list x, and the second argument is the permutation p.

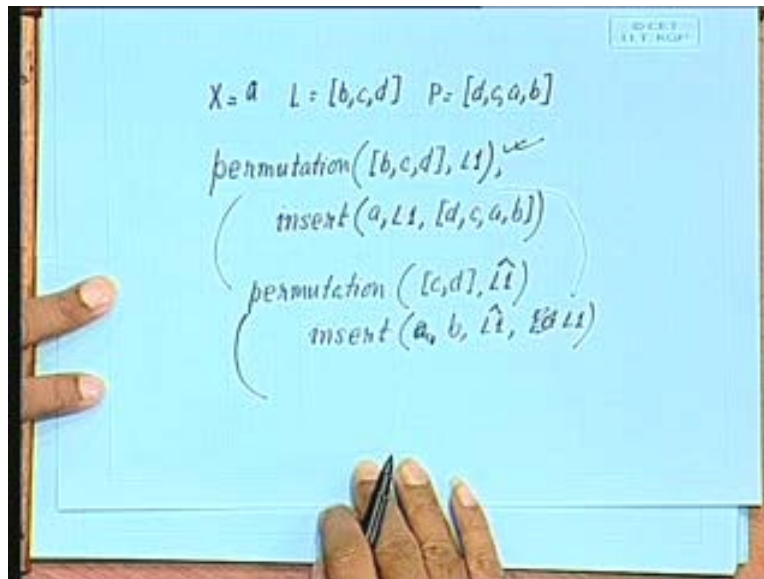
The first argument is this list- list argument- and the second argument is the permutation of the list. So, we say that this p is the permutation of x bar l, provided that l is the permutation of L1. What is happening? If you take L1, L1 is a permutation of this remaining list, and then, we can insert x at any point in L1, and then we will get p. Now, let us see again, with an example, that how does it work out, this thing? Suppose- (Student speaking). I will shortly define- insert is easy. Insert means, basically, that x has to be a member of p, and the remaining elements is going to be identical to L1.

In other words, if you delete x from p, you should get x one L1; if you delete x from p, you should get L1. Let us say that we are looking for the goal, where we are looking for permutation. If you look at the second rule here, then it is not going to match with the first rule, because the lists are not empty. It is going to try with the second rule, where it is going to instantiate x with a, and then, bcd is our L1, and our L1, is the list d c b, and when we insert x in L1 here, we get p, we get this. Basically, it finds out at which point we should insert this, so that we get this, and if there is no point where inserting this is going to produce this, then this is not a permutation of this.

Otherwise, this is a permutation of this. (Student speaking). See, at this point of time, it does not know what should be the value of L1, such that inserting x in L1 produces the permutation. It does not know, right? What it is going to do is, if you look at the way in which the binding will propagate, it will start with the final permutation that you have given. Because you have given it here, so it is going to start with this, and it is going to delete from this a, and so, it is going to find out dcb. (Student speaking). If you leave-yes. (Student speaking).

Which satisfies? No, that has to be checked again recursively. Recursively that has to be checked. (Student speaking). Which one? (Student speaking). Now see, that depends on how we have written insert. So, it is going to try to match- see, **what is the** it is going to go strictly by sub-goals. Let us see what are the sub-goals that it will generate. The moment you give something like this, it will first try to match this rule head with the rule head of your permutation program.

(Refer Slide Time: 00:41:38)



It will match x with a , it will match l with bcd . That is sufficient to unify the head, right? And it will match p with $dcab$. Now, what are the sub-goals that it has? It will create 2 sub-goals; one is permutation of l , which is bcd with $L1$. Now, what is $L1$? We do not know yet, right? And the second sub-goal will be insert x , which is a , again, in $L1$. We do not know yet, what is $L1$, but it should produce $dcab$. (Student speaking). We can write it also using the delete function. Now, what we do here is, when you have these 2 sub-goals, then, we can process these sub-goals in any order, right?

But what prolog will do is, prolog will go depth first, so it will first try with this one. When it goes depth first with this one, this is again going to generate 2 sub-goals; one is permutation of cd , to give some $L1$, because this is at the next level of this thing. This $L1$ and this $L1$ are not the same, right, and it will also generate the sub-goal insert, sorry, insert b , no, $L1$, right? From this level of recursion to this level of recursion, this is the binding that is getting carried down. Again, it is going to go depth first, so it will try this sub-goal first. In this way, it will continue until you hit upon the empty list, and then you will backtrack.

Try to just follow this sequence of instantiations and work out the complete derivation tree for permutation. That is going to give you enormous insight into the way in which the prolog will work, and I suggest that you try out with 2 scenarios. One is where your lists are already instantiated, so you are given 2 instantiated lists like this, and take another scenario, where the second one **is a var** is a variable x. In which case, prolog will print out all permutations of abcd? Is that all right? There is another way of writing permutation, and that is using delete, as you were asking this.

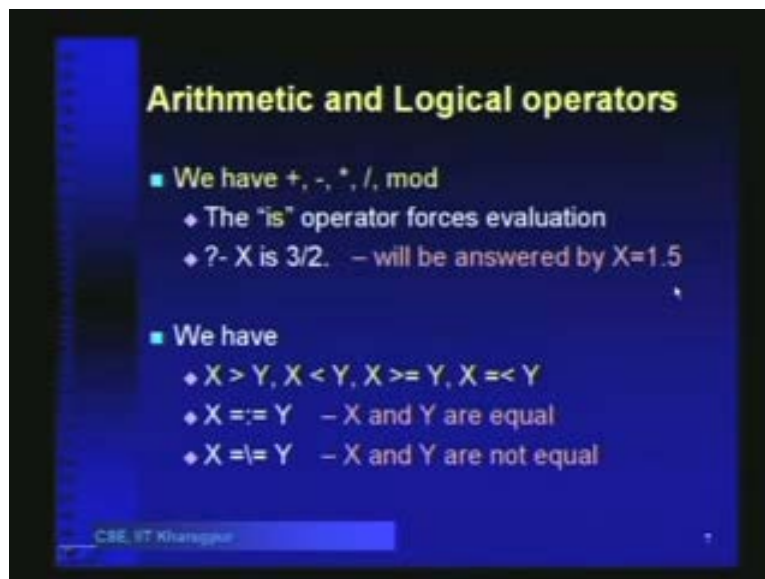
Now that you can write also in terms of delete, where you say that l is a permutation of x bar p, provided that by deleting from l, if we delete x, then we will get L1 and then L1 is a permutation of p. (Student speaking). Which order? (Student speaking). You can write it in that order. See, the problem that will happen here is that, if both are uninstantiated- I mean, if l is instantiated and this is uninstantiated, right, then p is not known at that point of time; when you try this one first, your L1 is not known, p is not known. So, you can potentially go into an infinite sequence of infinite recursion.

You need this one first, to get the binding of L1 **you to get** and a unification with L1, which you can carry up here; otherwise, your L1 and p will both be variables and you can go into an infinite recursion. **So that the the** The sequence in which you specify the rules and the sequence in which you specify the sub-goals is important in prolog, because prolog is not doing a resolution search, like we were doing in- when we worked out it out by hand. It is going to do a depth first search among the sub-goals. (Student speaking). No. (Student speaking). So, you want to treat delete as a function, which is going to return a list.

You can do that, but then you have to have predicates, which defines the delete function; that is also quite complex, that can also be done, but **but it is quite com** it is going to be quite complex and not very straight forward, because prolog supports functions, but we have to define those functions within predicates. (Student speaking). You can, but you have to also ensure that they do not appear more than once. (Student speaking). Yes. (Student speaking). Yes please, but see, member means what?

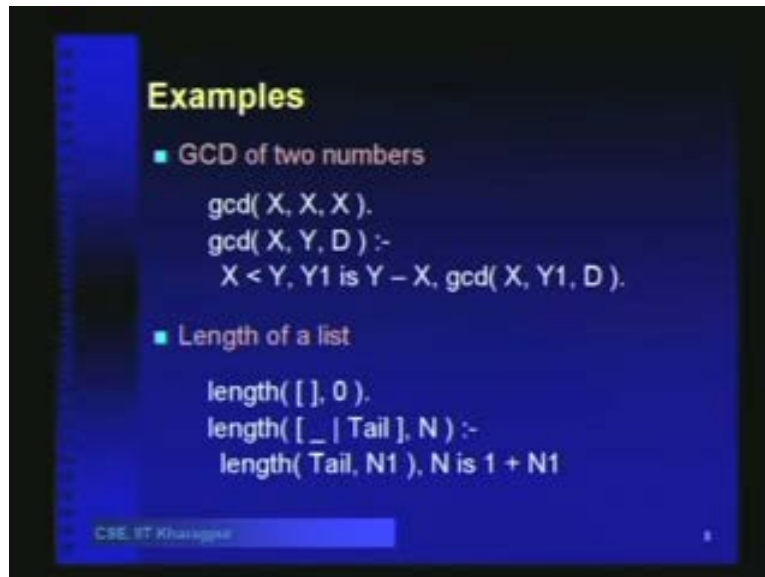
Suppose you have 3 as here, okay? I was referring to permutations, where you do not have repetitions, but if you had 3 as here and one a here, **then there is** then **this** each of these as is a member of this, **and each of** and this a is also a member of all 3 of them, but they are not permutations of each other- that is number 1. Even if you allow multiple permutation with duplications, then also, the number of as here and the number of as here has to be the same. Just by checking membership, we may not be able to do it so easily; we have to also check the number of occurrences with them. Had it been sets, then it would have been fine, but these are lists, not necessarily sets.

(Refer Slide Time: 00:47:53)



Prolog has some arithmetic and logical operators. We have plus, minus, star, division, mod, and the is operator forces evaluation. So, if we write that x is 3 by 2, that will be answered by x equal to 1.5. **It was** What it is going to do is- it will evaluate 3 by 2 and produce x equal to 1.5. We also have the following logical operators- we have greater than, less than, greater than equal to, less than equal to. Then we have equal and not equal, right?

(Refer Slide Time: 00:49:14)



We will see some examples with these. gcd of 2 numbers- gcd of x and y is z. The predicate is- gcd of x and y is z. If we have gcd x and x, that is x. **do you** Can you recognize this method? Yes. So, gcd of x and y is d, provided x is less than y. y one is y minus x and gcd of x and y one is d, right? **so its that its** It is a familiar recursive way of computing gcd, and that is been just coded into prolog. Length of a list: here, actually, what you should note is, in these 2- that we have used the less than operator here and we have used the is to evaluate **the um the** the value of y minus x here. Similarly, here, we are using the plus operator and the is operator to evaluate them.

So, what is happening here? We are checking the length of a list. Length of a list, which has some head, so this underscore here means that it is a wildcard- could be anything with tail, will have size n, is list, has size n, provided that the tail has size n1, and n is 1 plus n1, and the length of the empty list is 0. (Student speaking). This thing here- this underscore, which means you can replace it with any variable also, it just needs- is a wildcard. You can instantiate that with anything- x or head whatever, you can write it. This is a standard thing in prolog, actually. That **is all** is nothing special, just **another** like

another variable, right? (Student speaking). **If gc in in gcd if is** If your initial x is greater than y. Right. (Student speaking).

so in for For this to work, you have to basically put the smaller one **on the** as the first argument and the- (Student speaking). Yes. **yes yes um (Student speaking)**. I think we have to put another rule- yes. (Student speaking). Where the- you take care of the opposite point. I think we need to have another rule, which just interchange as x and y here, that will take care of the remaining part, is it? Not it. Oh, okay. Length of list is fine. Try to figure out, with an example, how it will work when you give a instantiated set of values. When you do not give any the value of the length, it should return the value of n, if the length is not given. So, we have a few more problems, but I guess we are running out of time, so we will do it in the next class.

I was hoping to finish prolog in this lecture, but okay, I will need one more lecture. (Student speaking). I think that has created lot of confusion- this underscore here, replace it by any variable of your choice. It is just some wildcard- that is all. You can replace it with x. It is anything which is not here, take that variable. We use that usually, when the value of that is not important, so it does not have to bind with anything else. Say, for example, if you put head here, then, in the rule body, **in the rule body** head is not binding with anything. So, it is something which is not important, so we just put this signal to indicate that it is a wildcard.

We do not need to bind that with anything. (Student speaking). If, in the left hand side, if you- (Student speaking). Yes. It means and of the sub-goals. (Student speaking). Or is between the choice of the rules. You can have more than one rule with the same rule head and that is the choice between the- it will first try the first one. If it fails in the first one, it will try the second; that is where the or is. So, **if you just** if you draw the entire derivation tree of prolog, you will get an and or graph, where the or is choosing which rule head to unify with, and once you unify with a particular rule head, then the set of sub-goals that you create is the set of- yes, is the and node. So, it is a pure and or graph that you will get as the derivation tree.