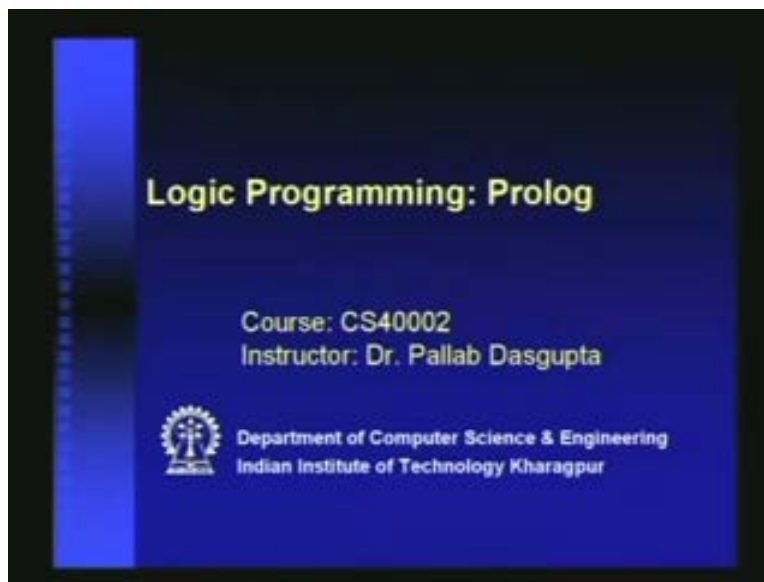


**Artificial Intelligence**  
**Prof. P. Dasgupta**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture- 13**  
**Logic Programming: Prolog**

Right. We will start a new kind of programming paradigm, called logic programming, with this. I mean, this is new for you, because you have not yet studied any other programming paradigm, except perhaps just a procedural paradigm.

(Refer Slide Time: 00:01:10)



**so logic prog** These logic programming languages are called declarative programming languages versus procedural programming languages like C, C++, etc. These are declarative, because you just have to specify in a declarative form, what you want to do without actually specifying how the program should execute. You are not specifying how the program should execute. You do not have things like if-then-else-for, which specify the control flow. Control flow is pre-specified. You only declare what you want to do. This is called a declarative programming style. We start by looking at some basics.

(Refer Slide Time: 00:13:10)

**Basics**

- 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 ) \*

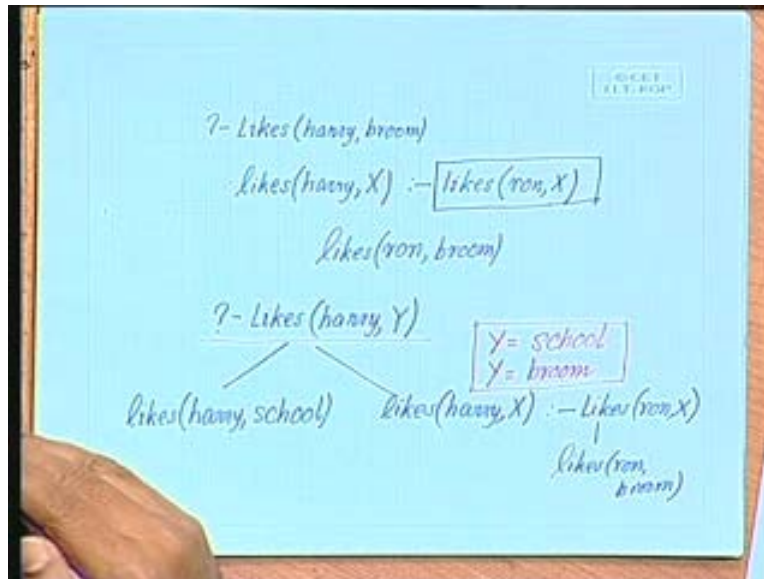
CSE, IIT Kharagpur

Firstly, we have the notion of instantiation, which is similar to first order logic. So, you can have predicates like likes Harry school, which is a fact, because Harry and school are given constants, likes Ron broom, and then, you can have rules of this form, where you have likes Harry x if likes Ron x. Here, in prolog, this is a sub-goal, and in order to satisfy this sub-goal, you have to satisfy these. In other words, the implication is from the right hand side to the left hand side. So, it says likes Ron x implies likes Harry x. If you have to deduce what Harry likes, then, to use this rule, you have to first deduce what Ron likes and we can have many other predicates on the right hand side.

So, you can have likes Ron x comma eats x something. We will see more examples. Now, let us see that what is the notion of instantiation here. We can instantiate the left hand side- the rule head with this predicate, right? You can have likes Harry school and you can have then likes Ron school- no, it is the other way around. From this, you should be able to deduce that likes Harry broom. Why? Because this is going to instantiate with this- right? Broom with x- and therefore, we will have likes Harry broom. Now, let us

consider some goals and see whether we can derive that. Can we deduce likes Harry broom? Yes? How do we do this? Let us see how prolog will do this.

(Refer Slide Time: 00:08:40)



It will start with likes- it will start with the goal. So, this is backward chaining- it is always going to do backward chaining, so it will start with likes Harry broom. This is how we specify a goal with a question mark followed by a dash. Then, it will try to see- what are the rule heads with this, which will unify with this? If you look at the rule- likes Harry x likes Ron x, this unifies with the rule here, with x substituted by broom. So now, the sub-goal becomes likes Ron broom. After applying this rule, the sub-goal becomes this, because we have unified with this, and we know that in order to get this, we have to get this.

So, the sub-problem is now of finding whether likes Ron broom, is supported by the knowledge base. And **in it is in um** it is indeed supported, because we have a fact here, which says likes Ron broom. Slides please, slides please. Yes, we have- likes Ron broom has 1 clause, so, therefore, we have been able to satisfy the sub-goal, and therefore, we can conclude that likes Harry broom is indeed correct. What happens if we give the goal

likes Harry y? Here, we have left a variable in the query. So, what prolog is going to do? It will find out all possible instantiations of y, which satisfies Harry y; satisfies likes Harry y. What is it going to do? It is again going to start by the instantiation, on likes Harry y, and then, it is going to unify this with these clauses 1 at a time. It will first pick up the clause likes Harry school.

This clause is there in the knowledge base, so it will unify y with school, and print out as the first result- it will print out school; it will print out y equal to school. And then, it will not stop there. It **will it** is also going to try out the other rules that exist in the knowledge base. So, it will try the next one. The next 1 is likes Harry x likes Ron x, and it will unify with this, create the sub-goal likes Ron x, and finally solve this by likes Ron broom. And therefore, the second thing that it will print is y equal to broom. So, this is going to be the output, if you give this as the goal.

Now, see this- it is important to note that- slides please, slides please- it is important to note that these clauses will be attempted in this particular order, so when you ask likes Harry y, it is first going to try this clause, then this, then this. If you have a recursion, then the base condition always has to be specified high up in the order, so that it first tries the base condition, then tries the recursion. We will see lot of examples of that. If you write it in the opposite way, it will- see, for example, if we put this clause- this likes Harry x if likes Ron x- if you put it ahead of these 2, then it is going to first try this one. And here, we do not have a recursion, because we have Harry here and we have Ron here, so it is not going to go into an infinite loop.

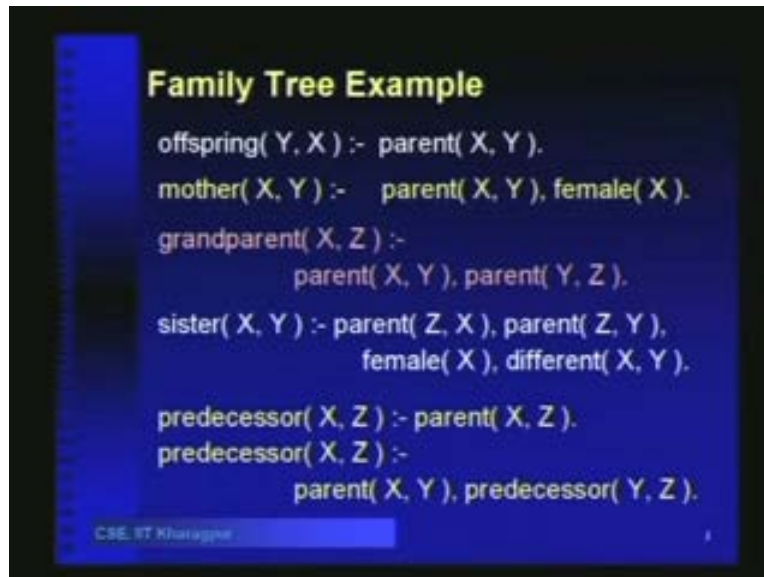
But we will see cases, where the order is going to determine whether you are able to terminate or not. And prolog is not going to reorder the clauses to ensure termination, it is just going to execute in that order. Let us look at a third goal. Suppose we ask likes z school. It is going to try instantiating this with a first clause, then, z will be replaced by Harry and it will output z equal to Harry. Then, it is going to try this- well, broom does not unify with school, so it will leave that, then, it will try this, and then, it will find that there is a substitution x with school and z with Harry.

So, it is going to look for likes Harry school, and then this side will become likes Ron school. Likes Ron school will become the sub-goal, but then, likes Ron school will not unify with this, it will not unify with this, it will not unify with this also, and therefore, we will deduce that likes Ron school is not correct. Now, note that because we are dealing with horn logic, we are always guaranteed to find both directions- the yes answer as well as the no answer. In this case, it finds out that likes Ron school is negative, right? So, the only thing that it is going to output here is z equal to Harry. Clear?

(Student speaking). No, I did not get your question. Yes, yes. So, if there is something- see, **it is** here is where as a programmer, your choice lies. If you want prolog to first checkout those clauses, you put them ahead of this clause. We will see more examples to see- I mean, where we should put the ground clauses? If you put them below the rules, then it is first going to check the rules and then go to the facts. So, usually, we will put all the facts first, so that prolog tries the facts first. It first checks whether it is already there in the knowledge base, and then it is going to try to use the rules to deduce sub-goals. We will go through several other examples, so, at the end of it. Now, suppose we ask likes zy, then, we are going to have- what is it going to do? It is first going to unify with this.

So, it will say z equal to Harry y equal to school- it is going to print pairs; z equal to Harry, y is equal to school. Then, it will print z equal to Ron, y equal to broom, right? Then, it will come here, unify z with Harry and x with y, and therefore, here also, we will have the sub-goal, likes Ron y, and when we have the sub-goal likes Ron y, then, we will instantiate with this, so y will be instantiated to broom. So, we will have z equal to Harry, y equal to broom. This is going to print 3 things- z equal to Harry, y equal to school, z equal to Ron, y equal to broom, and z equal to Harry, y equal to broom. Yes or no? Let us look at another example. Now, we are trying to see how to model different kind of relations in prolog. So, suppose we want to say that y is an offspring of x. Then, that is the same as saying that x is the parent of y. These are just examples. All of these are there in in the website.

(Refer Slide Time: 00:18:30)



You need not take them- **we are** these are all- just sit back and see. Mother xy indicates that x is the mother of y, so, we can write that as parent x is a parent of y and x is female, right? Grandparents- parent's parent, right? So, parent xy and parent yz. If x is a parent of z, then this can be satisfied, provided that we have a y, such that x is a parent of y and y is a parent of z; then, x is a grand parent of z. So, see, on this side, we can have many clauses, on the right side. If these clauses are true- if the and of these are true, then this is true. This is the horn logic that we were talking about. You have several clauses on the right hand side- several predicates on the right hand side- their conjunction implies what you have on the left hand side.

(Student speaking). No, we do not specify any quantifier here. The notion is that you have in this case- you have for all x, for all y, always outside. Yes, any variable that you have, you assume that are all universally quantified from outside. So, for all x, for all y, for all z, parent xy and parent yz implies grandparent xz, okay? So, that is, as were doing, you know, **this is already in clause form**, this is already in clause form, in the sense that **we have** we are writing it in the implicational form, but if you write it in not a or b style, then, we will see that it is already there in that form. So, that for all is always there, we

are not writing it after, then, your sister xy, parent zx, and parent zy and female x and different xy. different xy is important, because x and y can get instantiated to the same constant.

If x and y both get instantiated to Venus, and we have Venus is the sister of Venus, whereas we want Venus and Serena, right? This is clear? Predecessor xz- this is not complete, we have more of it here. Predecessor- **will use** we will have to use recursion. **So predecessor** Let us look at the rule first- the second one. Predecessor xz means the parent x is a parent of y and y is a predecessor of z, so, this is a recursion, and the basis of the recursion or the base condition is that pa- that it is- if x is a parent of z, then x is also a predecessor of z. If x is a parent of z, then x is a predecessor of z, otherwise, if x is a parent of y and y is a predecessor of z, then also, x is a predecessor of z. And we have to specify this in this order. If you specify it in the opposite order- if you specify this rule after this rule, then prolog will always try to do this and you will never term it.

If you want to check the predecessor of a and b; if you want to check predecessor ab, then you won't terminate. Is it clear? (Student speaking). We will write- this has 2 different rule- the first rule will be for all xz, for all x, for all z, parent xz implies predecessor xz. The second 1 will be written as: for all of x, for all y, for all z, parent xy and predecessor yz implies predecessor xz, whereas they have the same meaning. But, because in first order logic, you are **doing** going to do a search, you are not going to do just backward chaining; you are going to use full fledged search, right? Therefore, the ordering of the clause is not important, but prolog **is a** when well defined, it has a programming semantics.

So, it is going to always try this 1 first and the next 1 after that. It is going to always try in this order. It is never going to try something different, it is not going to- at some point of time, if it feels that it is in an infinite loop- let me try the other 1 first. Now, it is not going to do that. So, we have to understand the semantics of prolog and write the clauses in that order. Is it clear? Which one? No. If you want it that way, then you have to write it again, in the other direction also, right? So, here, we are writing that. So, what you want to say

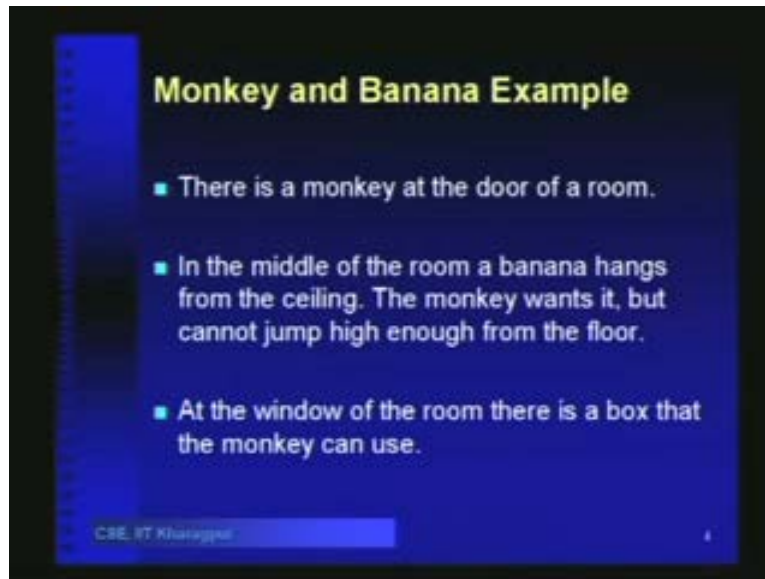
is, mother xy is parent xy and female x, and you want to say that this is the only way that mother xy can be added. Because you are not specifying any other rule, it is automatically implied. See, there is no other way that you can get mother xy, except by satisfying these 2 sub-goals, right?

So, the only part is there, because there is no other choice. On the other hand, here, you have 2 choices. It can be a direct parent or it can be parent's parent or parent's parent's parent. These 2 rules actually gives you more choice. Here, you have only 1 choice, so it is if and only if- that is what you are saying, right? (Student speaking). That is automatic, because it is not only if then you will have other rules. If you do not have other rules, it means it is only because if there is no other rule, then, there is no other way that you can derive that with the prolog engine. Let us [noise] we we look at a slightly more complicated example.

Here, I am I will try to show you how to use prolog to encode a search procedure. A search procedure, like the kind of procedure that we used for the missionaries and cannibals. That was essentially a way to plan the transport of the missionaries and the cannibals to the other side, and we frame that as a search problem, where we defined the state as the number of machineries, number of cannibals, and the position of the boat. We are going to see how we can encode such a state space within the prolog framework.



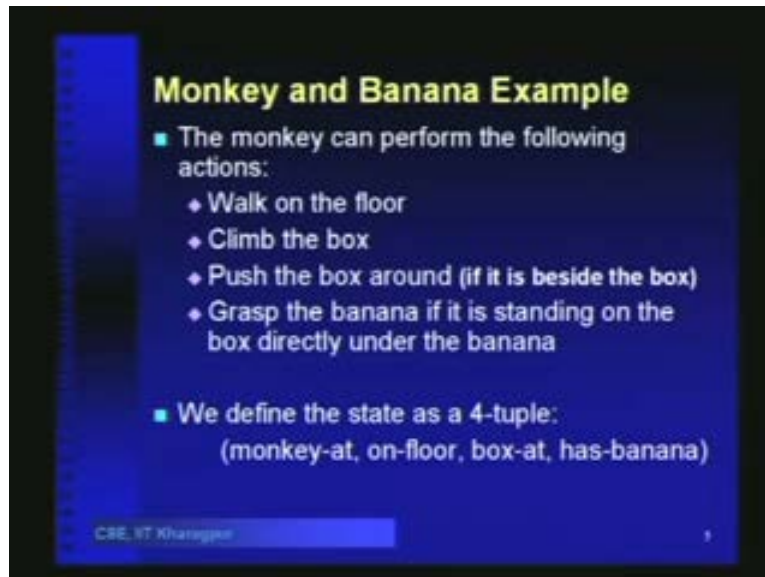
(Refer Slide Time: 00:24:41)



Then, we will go into other language features of prolog. I am just initially going to give you a glimpse of what we can do with prolog. So, the monkey and banana example- this is from the book of Russell and Norvig. This goes as follows- there is a monkey at the door of a room. In the middle of the room, a banana hangs from the ceiling. The monkey wants it, but it cannot jump high enough from the floor. Recall that the banana is in the middle of the room and the monkey is currently at the door of that room. Also, at the window of the room, there is a box that the monkey can use. It is actually besides the window and the monkey can push the box to the middle of the room, then climb on the box, and then it will be able to get hold of the banana.

The monkey can perform the following actions: it can walk on the floor, it can climb the box, it can push the box around, if it is besides the box and it can grasp the banana, if it is standing on the box directly under the banana. We will define the **state of what** state of the affairs with the four tuple, which gives us monkey at, which is the position of the monkey, right?- whether it is on floor or on box, then, the position of the box and **whether the monkey has been able to** whether it has the banana or not, right? This is the state.

(Refer Slide Time: 00:25:30)



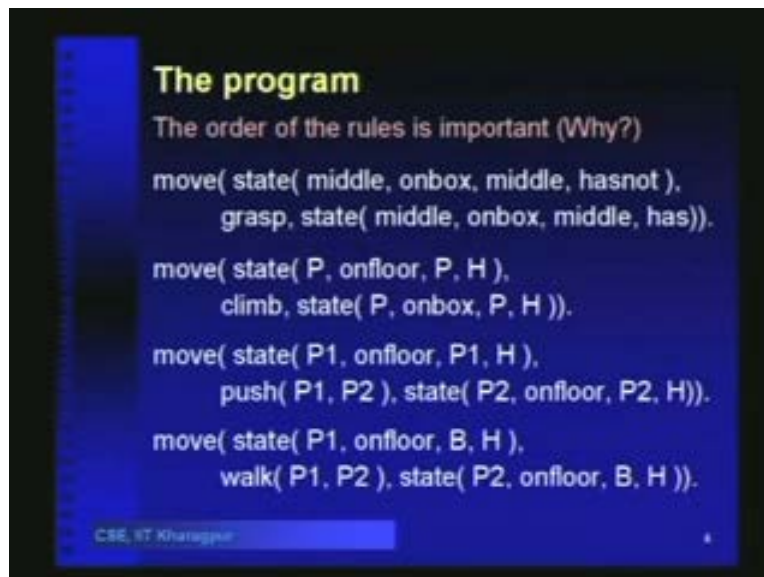
**Monkey and Banana Example**

- The monkey can perform the following actions:
  - ◆ Walk on the floor
  - ◆ Climb the box
  - ◆ Push the box around (if it is beside the box)
  - ◆ Grasp the banana if it is standing on the box directly under the banana
- We define the state as a 4-tuple:  
(monkey-at, on-floor, box-at, has-banana)

CSE, VT, Kharagpur

Now, like in first order logic, **we will** we are allowed to have functions in prolog. Let us look at the first predicate. The predicate move has 3 arguments **the predicate move has 3 arguments**. The first argument is the current state, the second argument is an action and the third argument is the final state. So, all the state transitions will be encoded in terms of the move predicate. All valid transitions- action transition pairs- will be encoded inside the move predicate. Let us see, that if we are in a state where the monkeys in the middle of the room, the monkey is on the box- the box is also in the middle of the room- and the monkey does not have the banana, then, if it uses the grasp action, then, the state will be middle on box; middle, and it will be able to grasp the banana.

(Refer Slide Time: 00:29:18)



Now, we put this clause at the top, because we always want prolog to first check whether we can satisfy this. And if you see, that this is the penultimate position of the monkey, it is at the middle, on box, box is in the middle, and it does not have the banana. So, this is the last step, so we put it at the first, so that we can always check whether this last step can be applied at every point of time. The **second rule the** second rule says, that if we are in, if the monkey is in a position p, it is on the floor; the box is also in p, and has or has not, whatever; this is a variable, so whatever. Then, by climbing, the state will become p; this onfloor will become onbox and the box will also remain in p and has or hasnot state will remain similar.

Third rule says, if we are in p1, and the monkey is on the floor, box is in p1, then by pushing from p1 to p2, the state becomes p2, onfloor box is also in p2, and h. Fourth rule: **if it** if the monkey is in p1, it is on the floor. Box can be anywhere and has or has not. Then, if it walks from p1 to p2, then the state becomes that the position p1 changes to p2, and the position of the box remains the same, because it has just walked; it has not pushed, right? So, these are the set of rules that we have to depict- the moves that the monkey can make, okay?

(Refer Slide Time: 00:31:20)



```
The program
canget( state( _, _, _, has ) ).

canget( State1 ) :-
    move( State1, Move, State2 ),
    canget( State2 ).

?- canget(
    state( atdoor, onfloor, atwindow, hasnot ) ).
```

CSE, IIT Khargpur 7

Next, we have this predicate canget. Let us see: the canget tells us that whether we can get the banana starting from state 1. We can get it starting from state 1, if there is a move which takes us from state 1 to state 2 with a move, and then canget state 2 is correct. Now, this move is a variable, mind you, **this is some** this move is a variable; if they just change of state from state 1 to state 2 and canget state 2, then we are done. And the ground condition is that whatever you have here, if you have has here, then your canget is true. Regardless of where you are, if you have been able to get the banana, **if you** if the banana state is has, then we have canget.

Finally, **the goal** the goal says, that can get from the state, atdoor monkeys, at the door, it is on the floor, box is atwindow and hasnot. Now try to see what happens, if we run this prolog program. If we run this prolog program with this has the goal, what is going to happen? The first thing that it is going to try is, it will try matching with this rule- the first one. See, it is going to try to match this goal with the rule head, so, it will try with this canget, then, it will see that it does not unify, because we have has here and we have

hasnot here. So, it is going to unify this state with state one, and so, we will have here: move state atdoor, onfloor, atwindow, hasnot,- some moves to state 2 and can get state 2. So, these will be our 2 sub-goals. Then, once it has this move, the first state is atdoor onfloor atwindow hasnot. It is going to try to unify that with these move predicates. It is not going to unify with this, because middle is not going to unify with atdoor. It is not going to unify with this, because this p and this p has to be identical, whereas in our current state, this is atdoor and this is atwindow. So, it will not unify with this rule head either. Similarly, it is not going to unify with this, because this also requires p1 and p1, but it will unify with this. We will have atdoor here, atwindow here and h here. Then, it can walk from p1 to p2; p2 is still not instantiated and state p2 onfloor b h. Now, what prolog is going to do is, it is going to try to instantiate p2 with all possible things, 1 by 1. So, it will keep this variable p2, fluid for the time being, and later on, when it again tries to unify this with another state, it will unify with atwindow.

And that is going to cause the monkey to walk from atdoor to atwindow. That is the first step. Then, similarly, by the similar reasoning, you will see that it will eventually move, push the box from window to middle. Then, it will be able to apply this rule, and it will be able to climb the box and then finally, it will be able to apply this and the final state is going to become has. When that becomes has, yes- when that becomes has, then this can get with the state 2 is going to unify with this has, and that is where we will terminate. We have some prolog interpreters in our laboratory. Please check out with the lab stuffs. I will inform them and try your hand at some prolog programs.

Because if we are to- you can try using that minesweeper problem, that assignment that I have given you. So, 1 option to do that is to use prolog to do the deduction for you, if you are able to write down the rules properly, so, please take a look at the prolog interpreter. Linux also comes with some prolog in its distribution. You can- if you have PCs in your rooms, then you can try out prolog there as well. Now, let us continue on this. We will now look at some more features of prolog. The prolog does not have data structures, it does not have any kinds of data structures like heap tree, etc., and that is actually drawback.

So, there has been developments to incorporate procedural predicates within prologs, so that you are able to write data structures in the procedural form, and just have predicates like insert, delete, etc., which is going to work on those data structures. The basic data structure **which has** which prolog supports is lists- it just has lists- and it can be lists of anything; symbolic lists. This can be written as follows: you can write lists as item 1 comma item 2 and so on. We have to enclose that within third brackets. You can have lists like head, then mid and tail. Now, this means that this head is the first item of the list and the remaining list is the tail.

(Refer Slide Time: 00:39:25)

**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

You can also write it in this head tail form, where head consists of several items, followed by the tail, which is the other item. For example, these lists are all equivalent for prolog- abc, or you can write a as the head and bc has the remaining list. See, what you have after the bar is another list, so, that is why, you see, we are enclosing that again in another pair of third brackets. This list in the head tail form, the first, is an item and then the second is the remaining list. So, you can have a comma b, and then, again, c is another list and then, you can have abc and the empty list at them. These are all identical

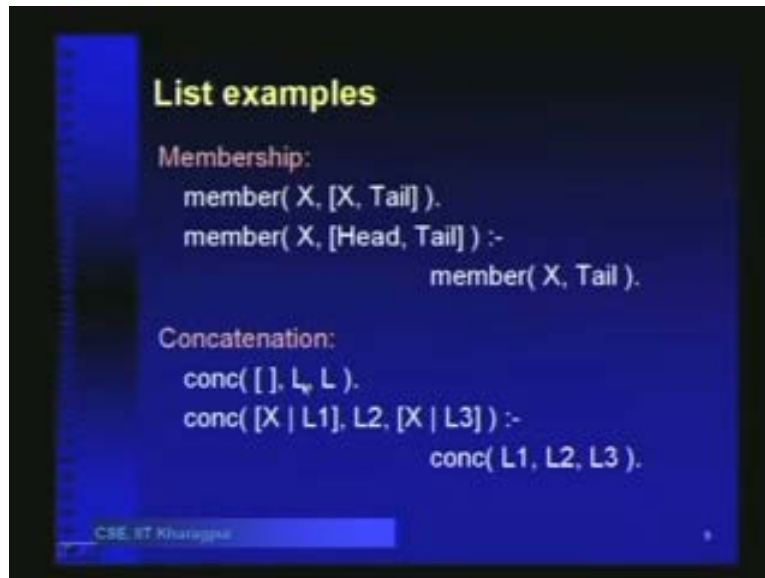
in prolog. Items can be lists as well. For example, see here: this list has 3 items- the first item is the list ab, the second item is c and the third item is a list, which again has d as the first item, and the list ef as the second item.

(Student speaking). No, this is not same as abcdef, right? This is a list of 3 items: first item is a list, second item is c, the third item is another list. The head of **the of** this list is the list ab, because I have this list as the head. Here, the head was a collection of 2 items, a comma b; here, it is a list, it is a single item- (Student speaking). The head, yes. **It is always** the head is always the first item of the list. Regardless of anything, head is the first item of the list. (Student speaking). Yes, yes, yes, but this this list of a comma b is the head. No, here a comma b is the head- a comma b is the head, that list is the head. Which one? **The** this 1 is the same, yes. No, no, no, this whole thing.

(Student speaking). No, if you if you do not put a third bracket here, then, head is a right. Yes, if you put a third bracket here, then the whole list is there. Yes, so here, head is a; here also, head is a, here also, head is a, here, head is the list ab. I got a little bit confused there. Now, is that is that alright? This slash. **this is** This tells us, these are items and this is the remaining list. Now, what we can have here is, we can have a variable which will instantiate with these lists. As we go further, you will see why require that slash, this is fine. (Student speaking). More than 1 means, you can have that nested within others, like you can have this as d. No, no, no, not more than one. No, if you have- it will be element, the least will be an element. After slash, you only have a list- whatever you have after the slash is a list.

Whatever you have before the slash can be a single item, like head, or it can be a collection of items. (Student speaking). No, that is going to be different, because then, the head is going to be the list a comma b, whereas in this case, the list is- the head is only a, is that right? Now, let us see the usage of lists. Suppose we want to deduce the membership of x in a list. So, this is how we will write it- the member predicate, where member xy, x is an item, y is a list and member xy will be true, if x is an item belonging to the list y.

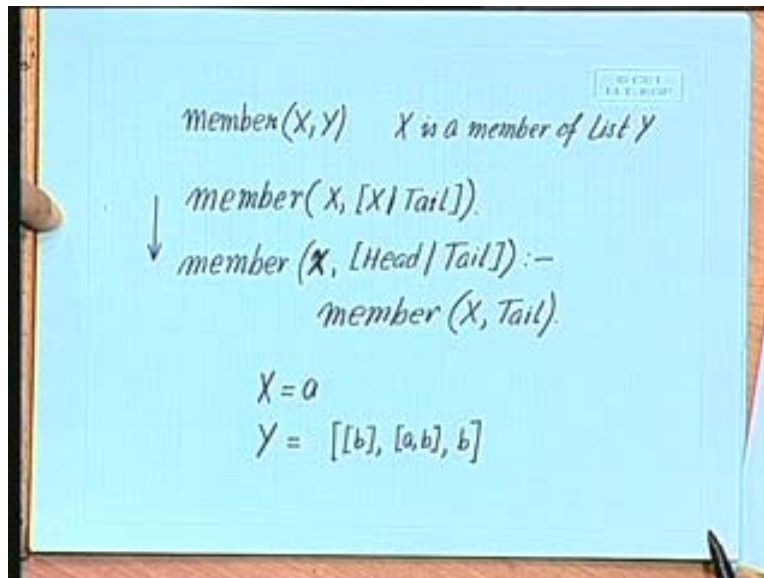
(Refer Slide Time: 00:51:53)



Here member x is- actually, I should have used mid here, I should have used the dash here. I thought I used a dash here, I do not know how it has become a comma. Anyway, just read this as- let me write it down. So, we are trying to write the predicate member xy, where x is a member of list y. So, if we find that this is the ground condition, where x matches with the first element of y, if we have member x, and then, if this is the first element, **then that is** then that is- then otherwise, what we need to do is, sorry, x, and then we have head tail, this is member. If this does not unify with this- if this unifies with this, then **we are** this will apply, **if this** because prolog is going to try in this order, so, we will come to this rule only when this head of the list- head of y- does not match with x. Tail is a list. No, tail is after the mid. What you have is, the tail- it is always list.



(Refer Slide Time: 00:47:31)

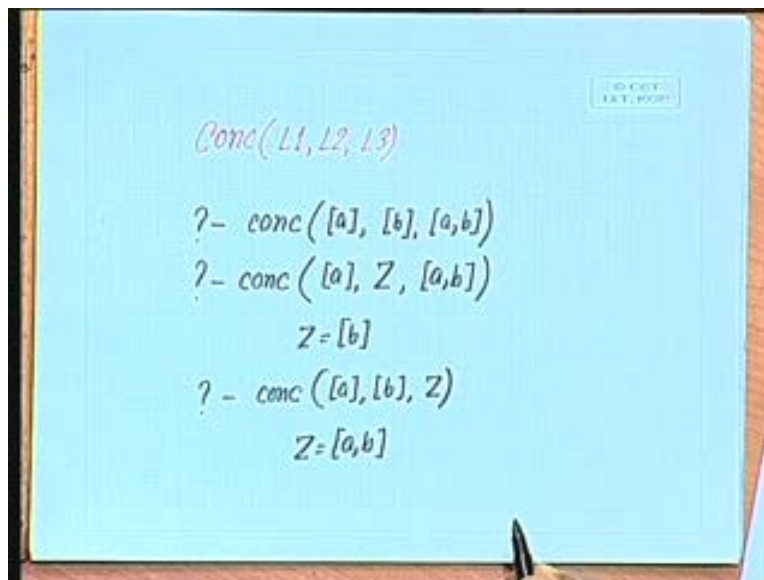


Then, we just need to check whether  $x$  is a member of tail, right? Here, we have just checked whether  $x$  is a member of the list  $y$ . Now, it is possible that  $x$  is not a member of list  $y$ , but  $x$  is a member,  $x$  belongs to some sub-list of  $y$ , because suppose we have  $y$ ; suppose  $x$  is  $a$  and  $y$  is the list. Now, when we apply this, what is going to happen? In the first,  $b$  will not unify with  $a$ , this list will not unify with  $a$ . So, we will try to check whether  $a$  is a member of this- of the tail- right? But then,  $a$  will not unify with this list. Again, this will also not match. The problem is that we are just looking at the list at 1 level of depth. So, if that does not unify with the head, we are just going to the next item, but even if it does not unify with head, it can unify with some sub-item of head, right? Take that down as an exercise, to find out the membership within sub-lists of  $y$ .

So, member  $xy$  will be true if  $x$  is a member of some sub-lists of  $y$ , which means that you have to recursively go down into the lists to the nestings of the lists. Previous slide- yes, say, even in these lists, if you are looking for item  $a$  with the membership function that we just now wrote, the membership predicate we wrote just now, we are not going to find it. (Student speaking). No, no, we are writing this. I am just showing you some glimpses

of predicate that we can write. There is nothing which is predefined- we have to write everything. This having a single element, yes, yes, yes. A singular lists- I mean, for example, here, when we wrote down- yes, yes, I mean, b within third brackets and b without third brackets are not the same, right? This is the membership function, where we just look at the first level of the list, not within the sub-lists.

(Refer Slide Time: 00:53:32)



Concatenation. Concatenation predicate is going to be like this, that concatenate- L1, L2, L3, right? So, concatenate L1 with L2 to get L3. Now, in prolog, that there is an equivalent between 2 things- 1 is that concatenate; you may look at in a procedure way. Concatenate L1 and L2 to get L3, or you can think of that given L1, L2, L3 has 3 different list variables, 3 different variables. This is true if L3 is the concatenation of L1 and L2. So remember, that it is a predicate- it has a true false value, right? So, if you concatenate the empty list- slides please. If you concatenate the empty list with l, we get l, right? So, if we have this empty and this l and this l, then that is always true, otherwise, if the first argument is x and then L1 second argument is L2 and third argument is x, followed by L3, then, this is true, provided that L1 and L2 concatenate and it gives us L3.

Now, see, you have to understand 1 thing, if you give 3 lists and **if your** if your query is given 3 list to deduce this thing, suppose I give a query, which says that conc a, b, ab, then, what is prolog going to return? It is going to return true, right? If you give conc a, z, ab, what is prolog going to return? It will return z equal to b, right? And if you give conc a, b, z, then it is going to return z equal to ab. So, this program that we have written here, is actually going to do all of these things. It is just a declarative way of describing in, using a recursion that what we mean by concatenation in first order logic. So, in, we will conclude this lecture now. In the next class, we will see some more examples on prolog, and then, I think I will give you some exercises to work on an actual prolog engine, to get a feel of what logic program in is like.