Artificial Intelligence: Logic Programming

Prof. Deepak Khemani

Department of Computer Science and Engineering

Indian Institute of Technology, Madras

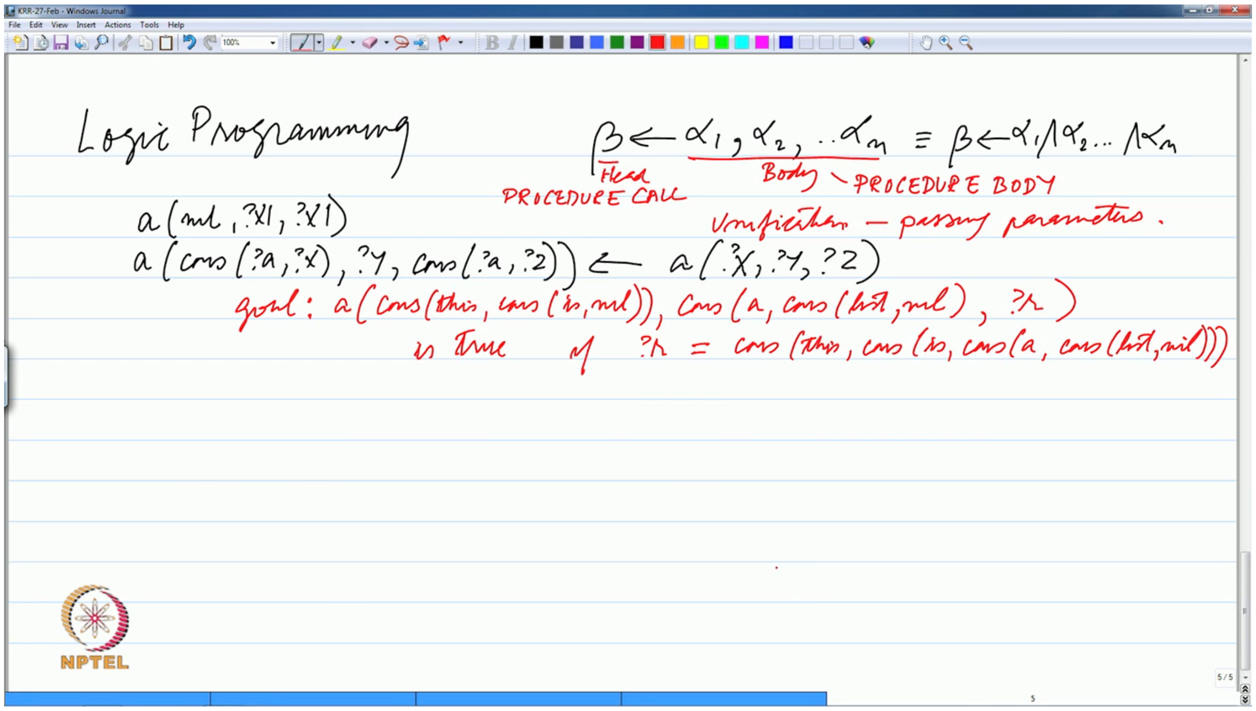
Module – 06

Lecture - 02

Okay so we are looking at this idea of logic programming. And if you look at a statement of the kind beta is true if alpha1 comma alpha 2 comma alpha n is true which is equivalent to saying beta is true is alpha 1 and alpha2 and alpha n is true but we will use the notation comma to stand for and as in done in the language prolog. Then one can think of the head so this is a head and so this is a head and this is a body. One can think of this body as a procedure body. And we can think of the head as a procedure call. If you are familiar with imperative programming and one can think of unification as passing parameters. So when by asking whether beta is true we are in some sense calling a procedure which in turn ask whether alpha 1 is true alpha2 is true alpha3 is true and so on and so forth. And which doing so it has to pass parameters which is done by the process of unification. So we are not looking at bigger formulas we have so far looked at smaller formulaes. And in the last class we looked at this definition of append which was just so let me just use a here. Nil x1 x1 and other one said that a cons a x y

And we discussed that how you can make a procedure call with a goal where you have so you can ask various questions here. You can ask were all the three arguments are specified or you can ask with any two specified and the third one unspecified. And that is what we did in the last class. We did try to show that this goal append cons this cons is nil comma cons a. that a goal of this kind is true if this variable we call it r right for result cons this cons is cons a.

(Refer Slide Time: 5:31)


Logic Programming

$$\underset{\text{Head}}{\beta} \leftarrow \underbrace{\alpha_1, \alpha_2, \ldots \alpha_n}_{\text{Body}} \equiv \beta \leftarrow \alpha_1 \wedge \alpha_2 \ldots \wedge \alpha_n$$

PROCEDURE CALL

PROCEDURE BODY

Unification – passing parameters.

a(nil, ?X1, ?X1)

a(cons(?a, ?X), ?Y, cons(?a, ?Z)) ← a(?X, ?Y, ?Z)

goal: a(cons(this, cons(is, nil)), cons(a, cons(list, nil)), ?r)

is true if ?r = cons(this, cons(is, cons(a, cons(list, nil))))

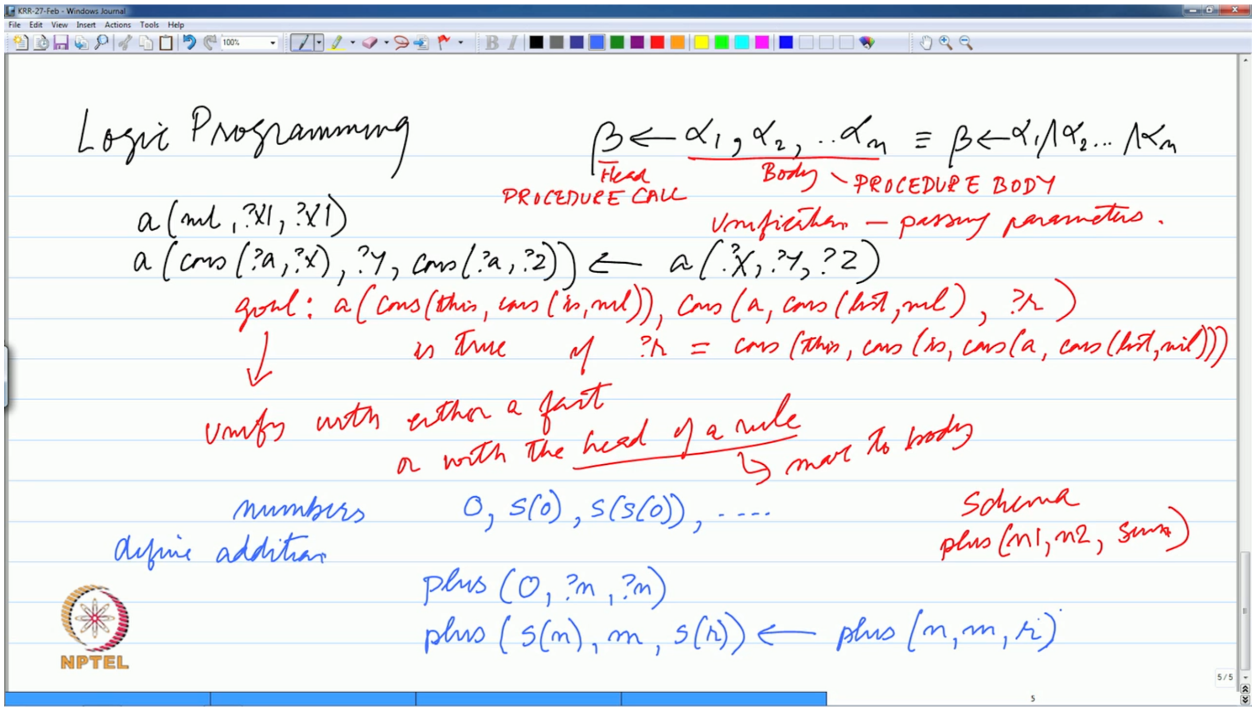
KRR-27-Feb - Windows Journal
File Edit View Insert Actions Tools Help
NPTEL

And we saw that this binding for this variable r can be obtained through a process of backward chaining. What is the process of backward chaining? That to show so you match the goal so actually I should use the term unify with either a fact or with the head of a rule. So it matches the fact then you are done it is true. If it matches the head of the rule then move to body. In our case ofcourse the body has only one formula so it makes life simple for us may be we will also pay some attention to what if the body had more than one formula what does our problem become like. Now just to sort of emphasize upon this lets look at logic program. And remember that we had this notion of natural numbers we had defined it at some point. We had said that a set of natural numbers is a set of 0 followed by successor of 0 which is 1 followed by successor of successor of 0 which is 2 and so on. We had defined this set of natural numbers. So just as we define the notion of append in logical sense lets also define an arithmetic operation which is operation of plus in the logical form. we will write it again in the same format.

So define addition. We want to be able to say what do you mean by addition logically. So 2 plus 3 is equal to 5 what is the meaning how do you arrive at that how do you say that. Now again very analogous to the definition of append so I am writing on the same page here we have a definition for addition and lets call this the predicate class and the schema of class is lets say number1 number2 and sum. The three inputs are numbers which satisfy our notion of what it means to add then the formula is true. So remember whenever we talk about logic we are talking about predicates and we are talking about when are formulas true. So we are saying a formula like plus 2 3 5 is true or a formula like plsy 3 7 10 is true but a formula like plus 3 4 9 is not true. So we are trying to separate true formulae from false formulae we are trying to find out which all formulae are true. And in the process we will define this notion of addition.

So how can we define addition. The base clause will be that if we add 0 to any number n you get the number n. and the recursive clause would be if you add the successor of any number to another number lets call it m and you get the successor of the result r. if this was true to start with n plus m gives r.

(Refer Slide Time: 10:11)



Logic Programming

$\beta \leftarrow \underbrace{\alpha_1, \alpha_2, \ldots \alpha_m}_{\text{Body}} \equiv \beta \leftarrow \alpha_1 \wedge \alpha_2 \ldots \wedge \alpha_m$

Head

PROCEDURE CALL

Body — PROCEDURE BODY

Unification — passing parameters.

a(nil, ?X1, ?X1)

a(cons(?a, ?X), ?Y, cons(?a, ?Z)) ← a(?X, ?Y, ?Z)

goal: a(cons(this, cons(is, nil)), cons(a, cons(list, nil)), ?r)

is true if ?r = cons(this, cons(is, cons(a, cons(list, nil))))

unify with either a fact

or with the head of a rule

↳ move to body

numbers 0, s(0), s(s(0)), ...

Schema

plus(n1, n2, sum)

define addition

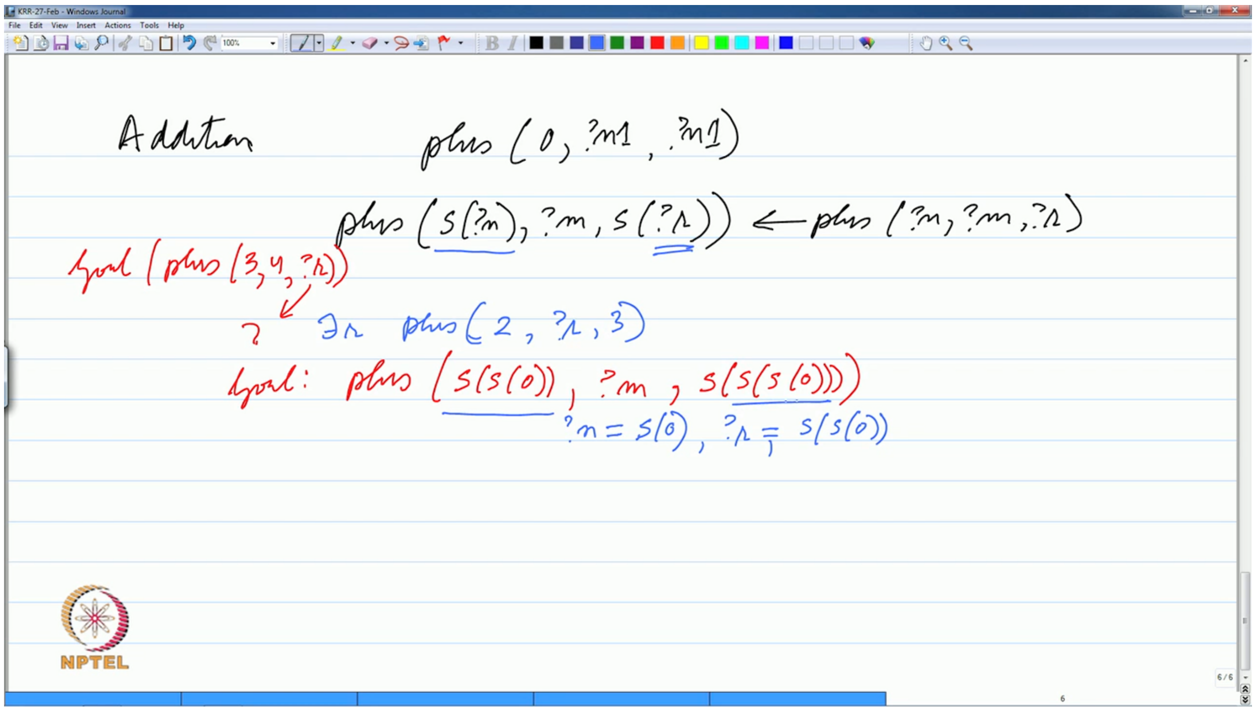
plus(0, ?n, ?n)

plus(s(n), m, s(r)) ← plus(n, m, r)

So if you read it in our original logical notation then you read it from right to left then you are saying is for all n for all m for all r if plus n m r is true then plus successor of n with m will give you successor of r is also true. So this is a program for adding two numbers. So lets see how this works. Just to use different variable names we emphasize on this. So If I give a goal as is this formula true lets say lets take another not very large number 2 plus 1 is equal to 3 then ofcourse you just plug it in. remember that 2 is successor of successor of 0 1 is successor of 0 you want to show that this formula is true. Let me finish that.so the goal should match the left hand side.

Now you can imagine if my goal was this plus 3 4 r I should really get 7 here. I wont go to the process but you should. You can see that we can even give goals as follows is this formula true. Plus so successor of successor of 0 with let us say lets talk of small numbers here so when I say goal I am basically asking remember its an existential query then does there exists an r such that this formula is true. Obviously in our human notation we are asking if plus 2 with some variable gives me 3. So there exists an r plus 2 gives 3 is it true that the formula is true. And ofcourse if you find out its 1 then its true. We can repeat this process that we did in the last class very quickly. You will match the first of the 3 arguments with the first of the 3 arguments here and this match will be done by n lets call it n lets give it a new variable name n2 and also call it. okay we don't need to do that here. R ofcourse remains there. I should have called this m. now r successor of successor of 0 because this r should match the remaining part of this.

(Refer Slide Time: 15:57)



Addition $\quad plus(0, ?n1, ?n1)$

$$plus(\underline{s(?n)}, ?m, s(\underline{?r})) \leftarrow plus(?n, ?m, ?r)$$

$goal(plus(3,4,?r))$

$? \quad \exists r \quad plus(2, ?r, 3)$

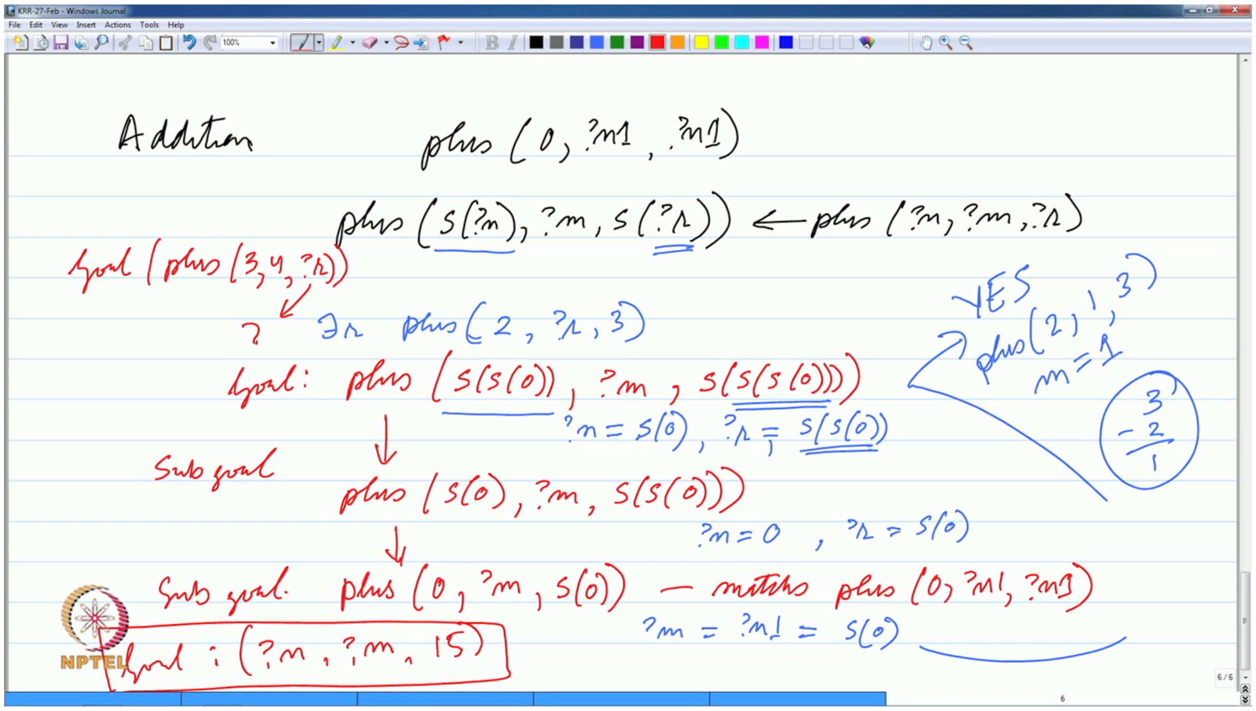
$goal: \ plus(\underline{s(s(0))}, ?m, s(\underline{s(s(0))}))$

$?n = s(0), \ ?r = s(s(0))$

So I get a subgoal plus successor of 0 m successor of successor of 0 so again I can now match. I say now n is equal to 0 and I can use same variable names here it doesn't really matter. From that I got this subgoal from this is got another subgoal which is plus 0 m successor of 0. Now this one matches plus 0 and what is the substitution that m is equal to n1 is equal to successor 0

So as a result of this we can answer yes and we can say plus 2 I will just write our notation now m is matched to 1 remember. Or we can say that m is equal to 1. What I have done I have subtracted 2 from 3. So what I have done is 3 minus 2 is equal to 1. This whole process I have done in this by saying that this is the variable I am looking for. The second of the two variables is my query. And now third if I use the third one then I would have been adding if I use the first or the second one then I am doing subtraction. So you can see that logic programming or atleast pure logic programming allows you to be very flexible. As long as you can define what do you mean by addition you can use the definition In many different ways. You can even give a query like plus are there two numbers n and m which add upto 15. And we will see that you can try running the program on this. Ofcourse this is very large numbers may be you can choose 3 or 4 instead of 15 and you can see that atleast get one answer. And then if you are working with prolog you will work with more than one answer. You can ask it to give the next answer and so on. So initially ofcourse you can see that it will match the first clause. Okay n is 0 and m is 15 and 0 plus 15 is 15 so the formula is true. Then if you say okay no give me another answer then it will go to the second clause and say that okay 1 plus 14 is 15 then the third answer is 2 plus 13 is 15. You can generate all the answers.

(Refer Slide Time: 19:57)



Addition plus(0, ?m1, ?m1)

plus(s(?m), ?m, s(?n)) ← plus(?m, ?m, ?n)

goal(plus(3, 4, ?n))

? $\exists n$ plus(2, ?n, 3)

goal: plus(s(s(0)), ?m, s(s(s(0))))

?m = s(0), ?n = s(s(0))

Sub goal plus(s(0), ?m, s(s(0)))

?m = 0, ?n = s(0)

Sub goal. plus(0, ?m, s(0)) — matches plus(0, ?m1, ?m1)

?m = ?m1 = s(0)

goal : (?m, ?m, 15)

YES plus(2, 1, 3) m = 1

$3 - 2 = 1$

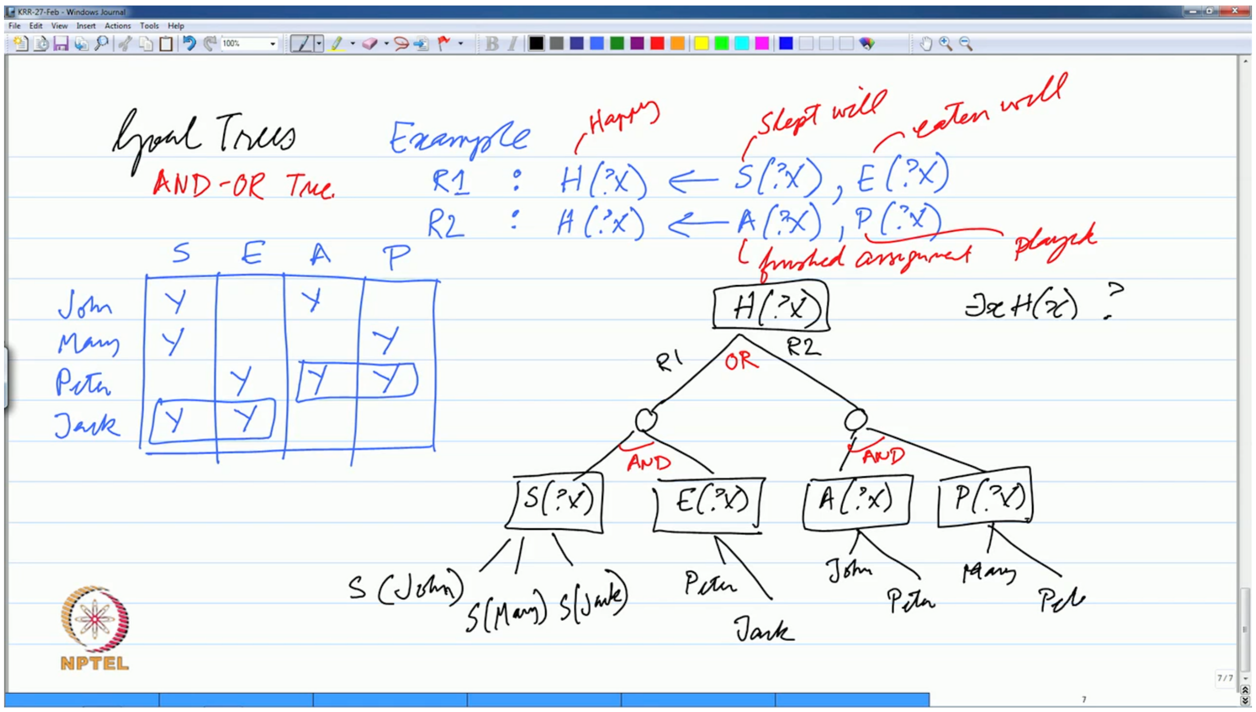
So there is lot of flexibility in logic programming. What if we have more than one we have so far the example that we have seen have no choices what if we have choices and what if we have larger antecedents. So lets take a small example to see what is the kind of thing we are talking about. Supposing and the thing I am trying to work towards is called Goal trees. We will see that essentially this kind of theorem proving is traversing through space which is called goal trees. So lets work with an example to make it simple. Lets say that we have two rules. Lets call one rule R1 this is just for our benefit. We are defining let us say what It means to be happy lets say. We say that H of X if let us say S of X E of X. we will give interpretations to these in a moment. But lets say we have two different rules. H of X is equal to A of X and P of X. it doesn’t really matter what these predicates mean but just to make it more accessible to us. Lets say H stands for happy and lets say S stands for slept well and lets say E stands for eaten well. Lets say this stands for finished assignment and lets say P stands for played in the evening. Doesn’t really matter.

So if you are given these two rules which are somebody’s definition of being happy and lets say we are given lots of facts as well. The facts are let me put it in the form of a table. Lets say we are talking about some people lets say john mary peter jack. And lets say we have some sort of a relational table. So this is for S this is for E this is for A and this is for P. lets say john has slept well and he has finished his assignment so we have Y for that or true for that if you want to say, Mary has also slept well and she has finished playing. Lets say peter has not slept well but he has eaten well and he has finished his assignment and he has also finished playing. And jack has slept well and eaten well. Lets say this is some data of some people we have. And then we ask the question is there somebody who is happy. And we are given this definition of being happy which says that if you have slept well and you have eaten well then you must be happy. The other one says that if you have finished your assignment and you have finished playing then you must be happy and we have this information about these four people and we need to ask. So if you look at the table you will see that there are indeed two cases which fit the notion of happiness. One is peter is happy because he has finished these two. And jack is also happy because he has finished these two. And because of the two rules we have. But how does backward chaining fit in here.

So the space that you want to explore remember that we want to move from head to body. Now we have two choices here that we could use the first rule which is sleeping and eating. Or second rule which is finishing the assignment and playing. Now as we see as we go along. Prolog would have first tried the first rule then second rule but lets first try to visualize the space of choices that we have. And that space can be seen as a goal tree. So the goal tree is as follows. That at the top level at the root you have the rule that we want to show to be true which is happy x. does there exist somebody who is happy. And remember we are doing this process of backward chaining. So you can either go to the body of the first rule or the second rule and let us say we stick to the lexical ordering we have. The first rule on the left and the second rule on the right and then we can really understand what prolog does. So this is R1 and this is R2 so we can either apply R1 or R2. So we come to a node here which we will give some name. lets call it R1 and R2. And then when you progress to the body of R1 we have two things to show. One is that so we have broken down the goal of happy x to the goal of eating well and sleeping well. That’s what rule R1 says.

The second rule in a similar fashion says that if we have done assignment and if we have finished playing then we could be happy. But now there is a difference in the three internal nodes that we have. The top level internal node is a choice and the choice is you can use R1 or you can use R2. But at the second level the unnamed nodes that we have that’s not a choice point that’s a node for which you have to visit both the children. And such nodes are called and nodes. And typically we put an arc across them to depict that they are and nodes. So such a tree is called an AND-OR tree and we have studied it. if you have done the AI course you have studied and or trees. Well what happens in backward chaining is the space that you generate is that of an and or tree. And then the data that we have we can just plug it in here. So john is here. So when I say write john here essentially I mean S john that john has slept well. That the formula S john is true and S Mary is true. And S jack is true. For the others I will just write the names do Peter is true and Jack is true for assignment john is true and peter is true and for playing mary is true and peter is true.

(Refer Slide Time: 29:00)



	S	E	A	P
John	Y		Y	
Mary	Y			Y
Peter		Y	Y	Y
Jack	Y	Y		

The solution of such a goal tree is a subtree where leaves are true and what is a subtree. It must take all children at and node. It must take all children at and node and one child at or node. Such a subtree is a solution tree. The leaves must be labeled with true formulae and at every and node you must have taken all the children and every or node you must have taken one of children. So as you can see there are two solutions here. One is shown in this red here. That there is a subtree that is satisfying this definition which is obtained by applying rule R1 then going down both the paths and then again that's an OR node here. All these are or nodes because we haven't put arc across them. So we have taken the path which leads to jack and we have got this arc. And you can see there is another solution which let me draw which blue here. Which goes along another path so you must go down both the paths and here you must go down peter and here you must go down.

So one solution contains peter here that peter has done assignments and peter has slept well. The other solution contains jack that jack has slept well and jack has eaten well. The task of the inference engine is to find one or more of this solution. The question that we have to ask is in what order shall we search this tree. And we will take this up in the next class and we will see prolog searches the tree in a depth first fashion which means it searches the left side first and then moves towards the right side. While you are doing this you Have to be careful that for example while you are trying to satisfy slept well and ate well or has eaten well the value of x must be the same. For example in this case if you can find that jack has slept well and jack has eaten well then you have a solution. That condition is there and that's the constraint imposed by the and node. And we will see that prolog does a depth first search on this tree which amounts to saying that it processes your formulae from top to down and left to right. So it will look at slept first and eaten afterwards because slept is on the left hand side and eaten is on the right hand side. And that strategy top to down left to right is a strategy which prolog uses and you can compare this with what ReteNet and OPS5 like languages did. There there were some problem solving strategy like specificity or recency which we talked about which decided which one to consider first. Now this is doing totally in static lexical order and as you will see this puts the onus on to the person who is writing the formulae. In writing them in proper order so that program works well. We have already started talking about it as a program and the person who is writing these formulae is the programmer and that leads us to this idea of logic programming.