Artificial Intelligence:

Logic Programming with Prolog

Prof. Deepak Khemani

Department of Computer Science and Engineering

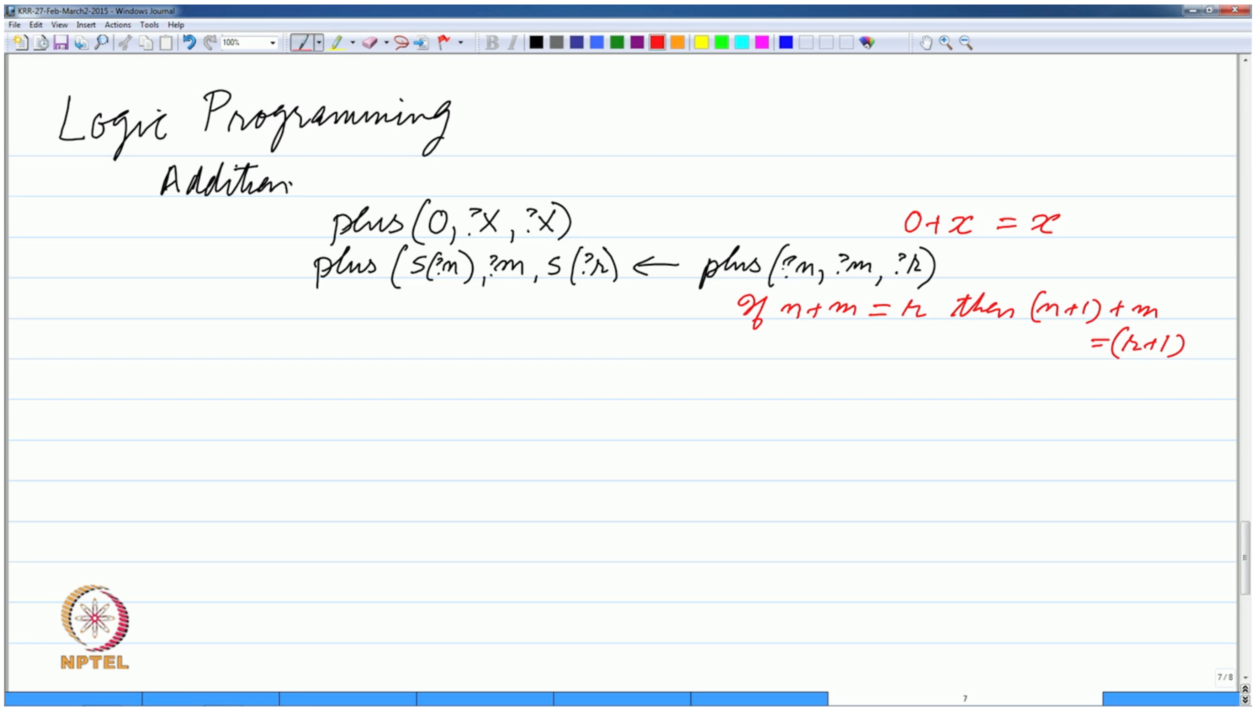
Indian Institute of Technology, Madras

Module – 06

Lecture - 03

Okay so we are looking at logic as a medium for programming or programming language. And we will call this process logic programming. And anything we can do in any other programming language can be also be done using logic programming language. In the last class we saw that we can define addition by saying and we are writing this in prolog style that if we have 0 added to any number you get the same number. So this is an axiom. And if you read it as a logic statement if you read it for all x 0 plus x is equal to x. and then we have the recursive clause which says we should really put question mark here because these are all variables. This basically says that if n plus m is equal to r then n plus 1 plus m should be equal to r plus 1 which is both are true statements so both are true statements logic is concerned with true statements.

(Refer Slide Time: 2:44)



Logic Programming

Addition

plus(0, ?X, ?X) $0+x = x$

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

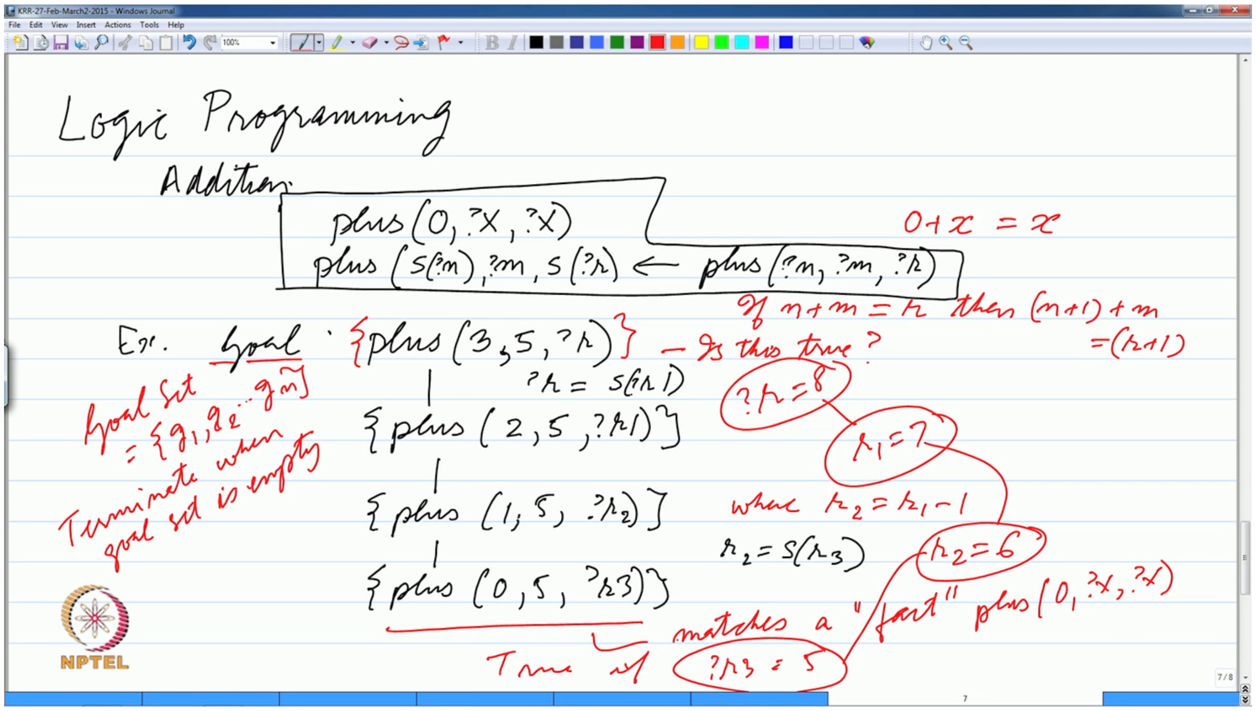
If $n+m = r$ then $(n+1)+m = (r+1)$

But what Kowalsky and other people showed was that by writing these true statements in an appropriate fashion you can actually write programs. We saw that these two statements that we have here is called program for adding two numbers.

As a small exercise you can having defined plus I would like you to define multiplication. So in fact this has been known for more than couple centuries that you can define addition in this fashion and then you can define multiplication as repetitive addition. And then we saw that if we have a goal like let me use numbers here. If this statement true plus 3 5 r . so the question we are asking is is this true because we are saying this is goal. And then a process of backward chaining we saw that we could reduce this goal so we will keep this goals in what we will call as the goal set. And we wil try to reduce each of these goals and we will stop. It mean if we don't have any more goals to solve then you have shown whatever you wanted to show. In our case we have only 1 goal and we have put that in the goal set and we are trying to show that this formula plus 3 5 and a variable how can this formula become true.

And we have seen that through backward chaining we will reduce it to a different goal which is 2 5 r1 where r is equal to successor of r1. And this in turn we will reduce to plus 1 5 r2 where r2 is again or you can write it as follows where r2 is r1 minus 1. In other words r1 is the successor of r2. And then we reduce it to a goal plus 0 5 r3 where again r2 is equal to successor of r3. I am not writing the question marks here. But at this place we don't need to reduce it to a subgoal because this matches a what we can call as fact which is that plus 0 anything is. So all we have to do is that if we say r3 is equal to 5 then this is true.   So once we know that r3 is equal to 5 is true we know that r2 is equal to 6 once we know that we know that r1 is equal to 7 and therefore we know that r is equal to 8. Simply through a process of substitution.

(Refer Slide Time: 7:47)


Logic Programming

Addition

$$\boxed{\begin{array}{l} plus(0, ?X, ?X) \\ plus(s(?n), ?m, s(?r)) \leftarrow plus(?n, ?m, ?r) \end{array}}$$

$0 + x = x$

If $n + m = r$ then $(n+1) + m = (r+1)$

Ex. Goal : $\{plus(3, 5, ?r)\}$ — Is this true?

$?r = s(?r1)$

$\{plus(2, 5, ?r1)\}$

$\{plus(1, 5, ?r_2)\}$ where $r_2 = r_1 - 1$

$r_2 = s(r_3)$

$\{plus(0, 5, ?r3)\}$

True if $?r3 = 5$ matches a "fact" $plus(0, ?x, ?x)$

$?r = 8$, $r_1 = 7$, $r_2 = 6$

Goal Set $= \{g_1, g_2 \ldots g_m\}$

Terminate when goal set is empty

We know that this formula is true the last formula in this sequence if r3 were to match this which means successor r2 must match 6 and whose successor r1 must match 7 and whose successor r must match 8 and the original formula is true if r matches 8. Here we can return the answer yes the statement is true provided r is equal to 8 and in other words what you have done is we have added two numbers 3 plus 5 to give us r. so this is basically showing that addition has a logical foundation. In practice you can imagine that we are not going to do addition like this because it’s a very painstaking process. And you can see here that the complexity the number of recursive calls you have to make is proportional to the size of the first number that you are trying to add which means that if you wanted to add 1000 plus 20 you would end up making 1000 calls to this program before you got the sum.

And obviously we don’t think addition as such a complicated process. And in practice even a prolog program will also not do addition like this. But this is just to show that logically we know how to add tw numbers. So lets do some more arithmetic. And lets look at this sequence called Fibonacci numbers. So you must be familiar with this sequence you start with 1 then 1 then you keep adding the last two numbers to give you the next number. So 2 and 1 plus 2 is 3 and 2 plus 3 is 5 3 plus 5 is 8. 8 plus 5 is 13 and so on. So such a sequence is called as you know a Fibonacci sequence. And it is fairly straightforward to translate the definition of Fibonacci numbers into a program to compute Fibonacci numbers. So a simple program as you can imagine would be as follows. That we define that lets say this is a this thing.

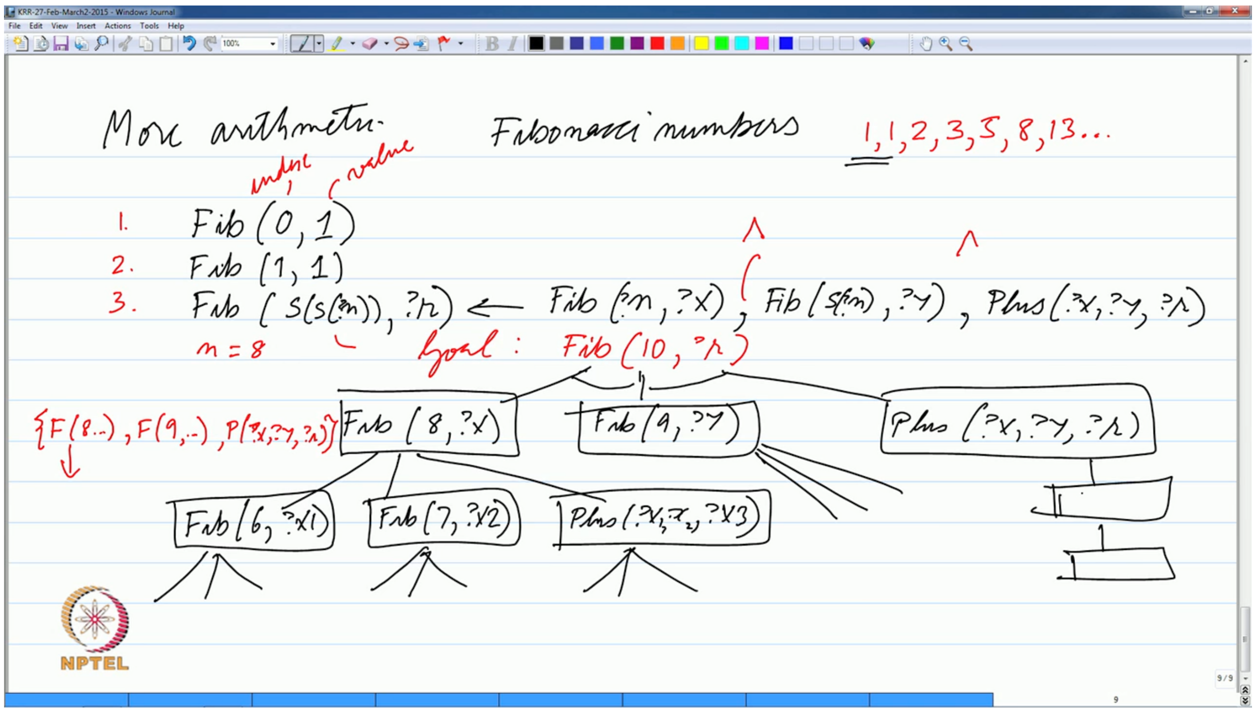
Lets say 0 is the index of the number and 1 is the value of the number. So the zeroth number of if you want to say first number it doesn’t really matter is 1 then the next number is also 1 because we start with these two numbers. The rest is defined recursively. And I m sure you can write this quite easily now so successor of successor of n. so what is n here n means we want the n th Fibonacci number. In fact we want n plus second Fibonacci number in this case because it is a successor or successor of n. will be true if we know the n Fibonacci number sorry I should have is a r the result if I know the previous Fibonacci number or two before. The n th Fibonacci number is lets say X and the next Fibonacci number is n plus 1 Fibonacci number is y and now we can use our program to add two numbers X Y r.

Observe that this is an and sign here. In prolog style we are putting a comma but if have to read this as a logic statement you want to say that for all x for all y for all r and for all n even n is a variable, if n th fibonacci number is x and n plus 1 fibonacci number is y and if x plus y is equal to r then n plus second fibonacci number is r. so that’s the definition of fibonacci numbers. So if you wanted to solve for example a goal that calculate for example the 10 th fibonacci number we would state it as follows so I will just write 10 here and I will take it that you understand that basically there are 10 s’s in a row followed by 0. Is r .if that is my goal then we have seen it in last class that we end up exploring a tree called a goal tree which means that to show that this is true I will have to show three facts which are the body of the above rule.

So either I match it with either the first statement or the second statement or the third statement. And we have already mentioned that prolog will try this in the order in which we write them. Now obviously this goal fibonacci 10 tenth number will not match the first statement because that's talking of the 0 th fibonacci number. Will not match the second statement which is talking about second fibonacci number. So it will match the it will match this one with a value that n is equal to 8 because when n is equal to 8 successor of 8 is 9 successor of 9 is 10 so we are talking about the number 10. Now we will break this down into the three subgoals which are the body of this particular rule which is that fib 8 x fib 9 y plus. And as we had mentioned earlier it is a and node in the sense that you have to follow all three. You will also follow this notation that I just introduced. That we will write this a goalset which will say this fibonacci for 8 I am not filling in the details. fibonacci for 9 and this plus of x and y r. three goals we put in the goal set and again prolog will take the first one which amounts to doing a depth first search on this goal tree we are talking about.

How will this go. To show this let me just okay lets start as prolog would do it. it would now break up this goal it will try to show this. First it will try to match it with the first statement but it wont match because the first statement is talking about the 0 th number it wont match the second likewise It will match the third one and we will have what you can see is the recursive call for fibonacci 6. Lets use some variable x1 here fibonacci 7 x2 let me call it doesn't matter some variable name. plus x1 x 2 x3. And then It will do this. Three places it will do this. Likewise its gonna do this one. The right hand side is the simplest because it has as you have seen a linear structure. We will keep reducing x so x plus y is r x minus 1 plus y is r minus 1 x minus 2 plus y is r minus 2.

(Refer Slide Time: 17:41)



So let me just put one or two more here. So this one for example. This is talking about the 7 th number and this is talking about the 8 th number and this is ofcourse plus. Now the point is as you can see there is a lot of wasted working going on here. That you are trying to show that fibonacci you are trying to show what is 7 th fibonacci number here as well as here. And likewise you are trying to compute the 8 th Fibonacci number here as well as here. And there is a lot of repetition going on here. And this one grows exponentially. Every node has 3 children and you will have to as you can see go at least 10 level down. Ofcourse some of them will taper off 6 will reduce to 4 as its leftmost child which will reduce to 2 as its leftmost child which will reduce to 0 as its leftmost child but that will terminate because it will match. As some point it will terminate but it's a big tree and computes does a lot of unnecessary computation.

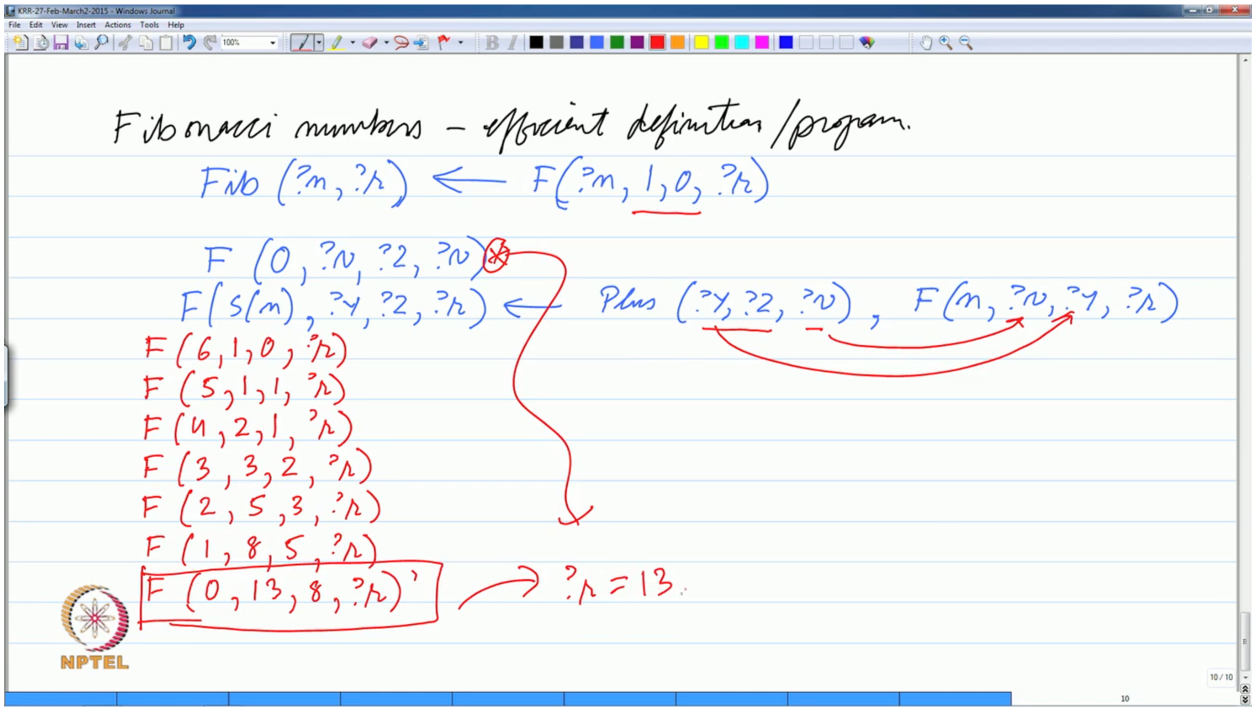
So what we are trying to show is that even though you are trying to think of logic as a programming language and you have a straightforward way of translating a definition like fibonacci numbers into a program it doesn't make sense to do that because computationally it is very inefficient. So lets look at how we can do it better. So we will think of statements both as logic as well as programs. So here is an alternative way of doing this and this is a practice we often do in many programming languages in that we introduce new variables and try to convert some of these recursive programs into iterative programs. So lets call this as a function f or predicate f rather which is talking about this and we introduce two values 1 0 and the result r. so an auxiliary program we are writing by adding new variables. We saw this we did this when we were looking at unification algorithms we had an auxiliary program there the substitution was being built up. And this auxiliary function the base clause is that when you are talking about the 0 th number then whatever is the value here and whatever is the value here must be the value here.

So we are going to recurse on n as before and this and when n becomes 0 this particular thing will hold. Otherwise if you are looking and n plus first fibonacci number

Okay so just to give you an intuition behind what is happening here is that we start with these two pairs of numbers 1 and 0 and then we keep adding them and keep so we add y plus z to give v and then we bring this v here we bring the larger of the two numbers here. So we are shifting them. Lets do an example and that will make things clear.

So lets say I am doing fibonacci of 6 and I make a call with 1 0 and r. so I will match this with this and move to the right hand side. So ignore the plus sign we will assume you do plus eventually. So what is plus going to do. 1 plus 0 is equal to 1 it will give us provided the value of v will be 1 so we will get at the next level fibonacci of 5 now 1 will come here and r will remain there. Again we can the same rule. We will go to fibonacci 4 so 1 plus 1 is equal to 2 will come here 1 will move here and r will come here. Then we will call fibonacci 3. Now 2 plus 1 3 will come here and larger of those two numbers 2 will remain here and that is the result we have been looking for. Then we will go to the call with 2 so 3 plus 2 5 will come here and the larger of them will go there. 5 plus 3 8 and 5 will come here. 8 plus 5 13 8 will come here. But at this statge this one we will not need to recurse anymore because this will match this clause and we will get r is equal to 13.

(Refer Slide Time: 25:49)


KRR-27-Feb-March2-2015 - Windows Journal
File Edit View Insert Actions Tools Help
NPTEL

Fibonacci numbers – efficient definition/program.

$Fib(?n, ?r) \leftarrow F(?n, 1, 0, ?r)$

$F(0, ?v, ?z, ?v)$

$F(s(n), ?y, ?z, ?r) \leftarrow Plus(?y, ?z, ?v), F(n, ?v, ?y, ?r)$

$F(6,1,0,?r)$

$F(5,1,1,?r)$

$F(4,2,1,?r)$

$F(3,3,2,?r)$

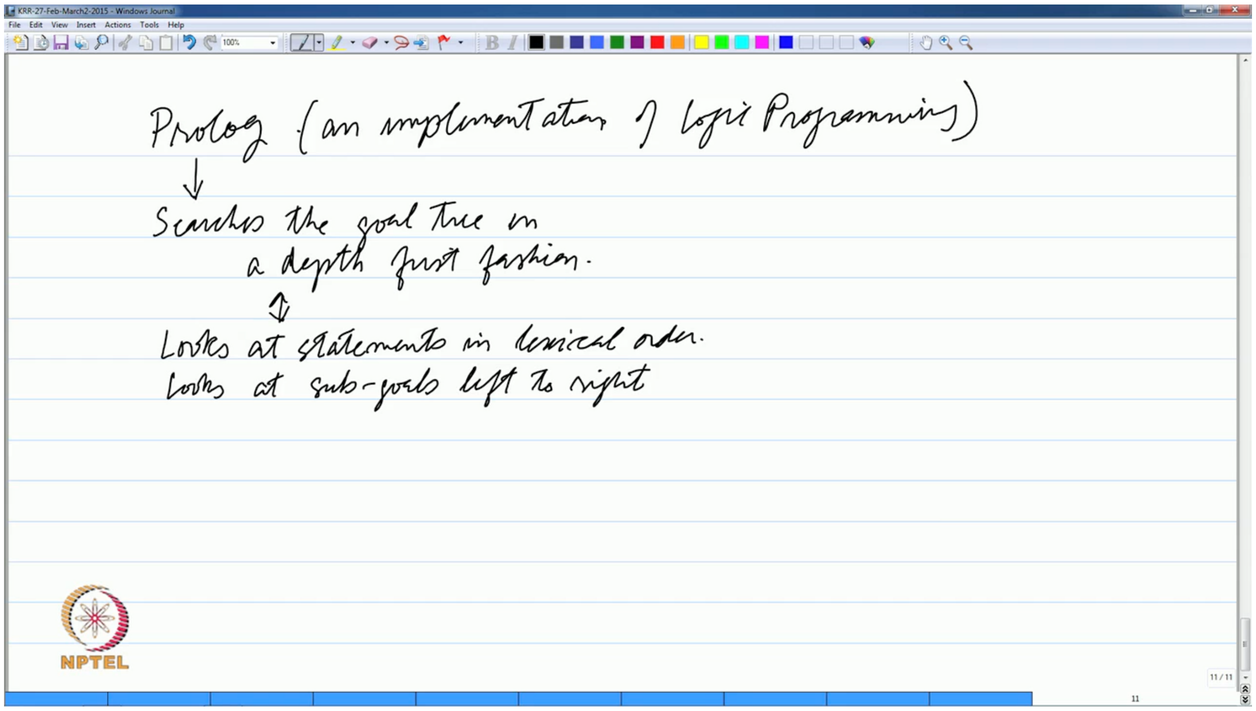
$F(2,5,3,?r)$

$F(1,8,5,?r)$

$\boxed{F(0,13,8,?r)} \rightarrow ?r = 13$

So as long as argument was positive 6 5 4 3 2 1 we made a recursive call and we reduced it to smaller problems. The moment it became 0 we had results stored in the variable r and we can return he value.now you can see that first of all this is linear as opposed to exponential. So if you are using logic as a programming language we still need to worry about writing the clauses quite efficiently. Now prolog is an implementation of logic programming. It is one implementation of logic programming it doesn't have to be. What prolog does is that searches the goal tree in a depth first fashion. Which translates to saying that looks at statements in lexical order and looks at subgoals left to right.

(Refer Slide Time: 28:41)



Prolog (an implementation of Logic Programming)

$\downarrow$

Searches the goal tree in a depth first fashion.

$\Updownarrow$

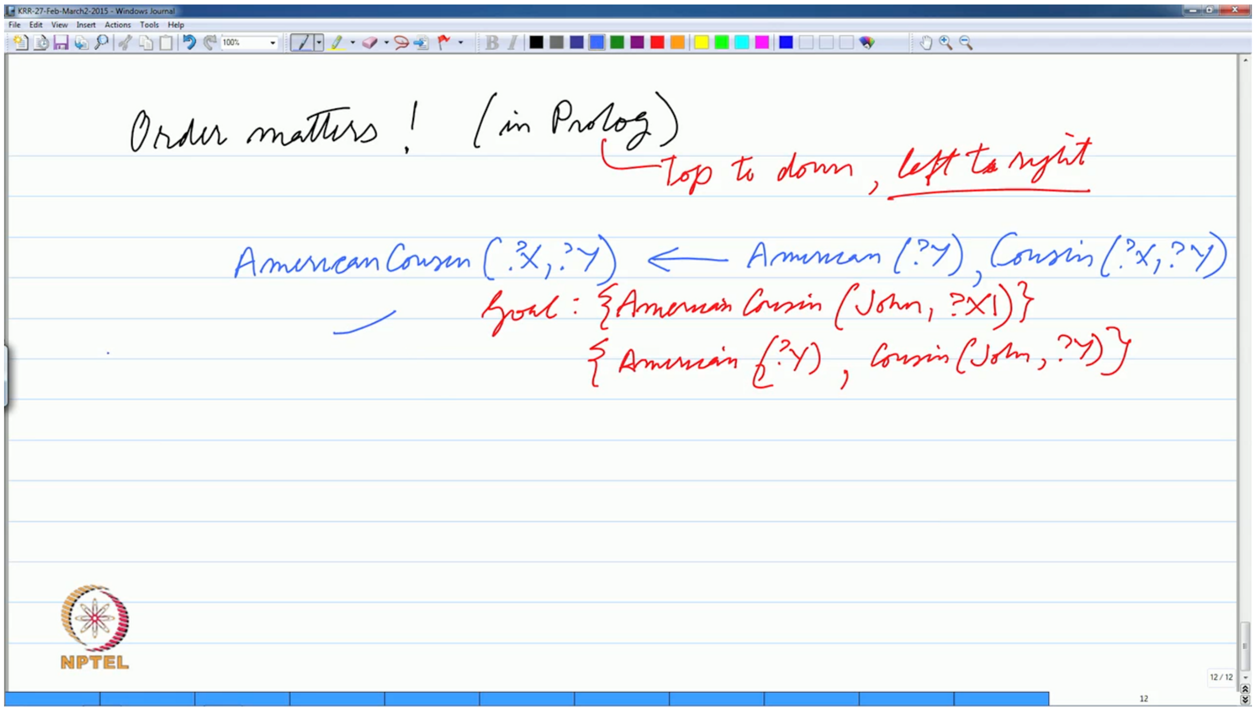
Looks at statements in lexical order.

Looks at sub-goals left to right

So the order in which you write matters. It critically influences the program will work. So obviously one thing we have seen is that base clauses first when you are doing recursion. Because everytime it is trying a goal it must first the base clause and then must move to the recursive call. To write them in different order then they would do things differently. So the thing is that order matters. Now in the logic programming in pure logic programing one assumes non deterministic machine. So Kowalski's idea was the following that a program equal to logic plus control. Now this control in prolog it is depth first search. We already stated that prolog looks at things in depth first search. But ideally this control would be non deterministic in pure language programming. A user need only state the logic part. The rest should be left to the machine and when we say machine we mean the inference engine. But that's only in ideal case in the pure logic programming you don't worry about in what order you write things. Because you treat all the sentences as a set and the inference engine somehow picks the right things and does things. Whereas in practice when you use a language like prolog you have to write things in a specific order.

So lets look at a example. So we had said that prolog works top to down and left to right. Lets look at this left to right portion lets look at an example. Supposing you are trying to define a predicate called American Cousin and you define it as follows that X so the meaning of this is y is is American cousin of x provided two things. One thing is y is the cousin of x and second things is y is American. So there are two subgoals that you are looking at. So supposing you write it like this. What is this program what will this do. That if I have to ask the question lets say goal American cousin john x lets say x1. So we are asking does john have an American cousin. What will this particular program do it will say it will break it down into goal set so starting with this goalset we will reduce it to this goal set. American y and cousin. What is this saying.

(Refer Slide Time: 35:13)



Order matters! (in Prolog)

Top to down, left to right

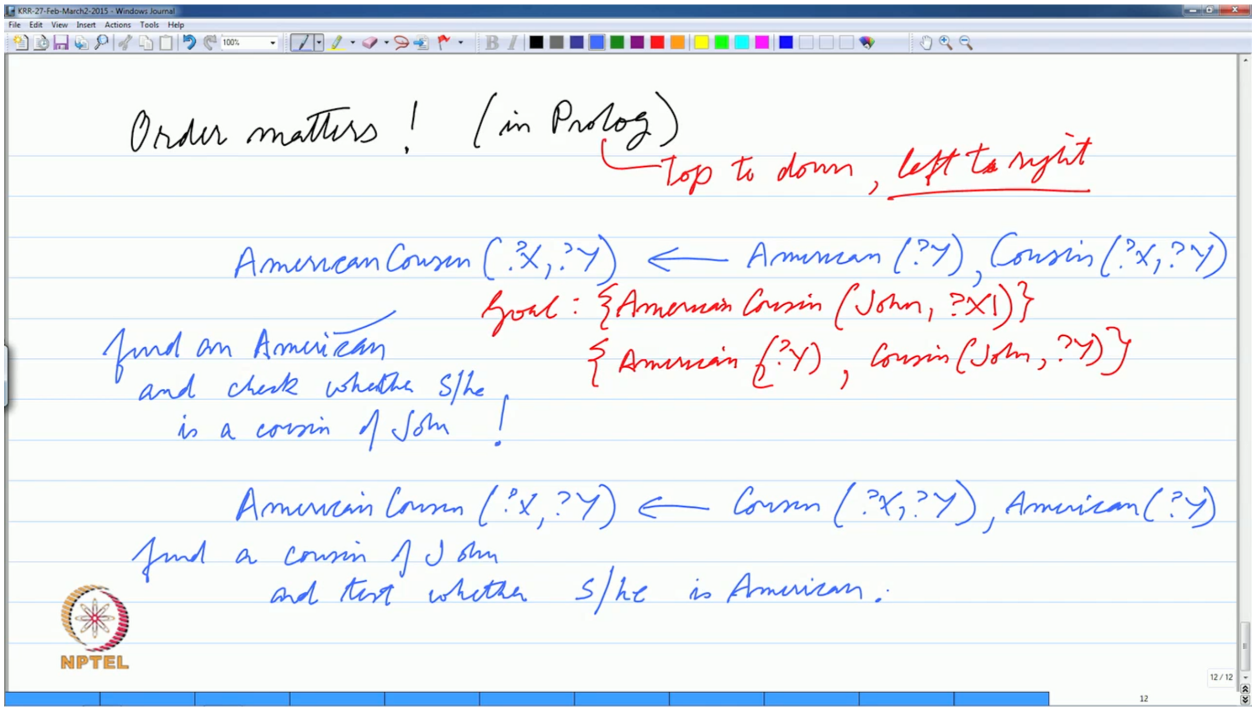
AmericanCousin(?X, ?Y) $\leftarrow$ American(?Y), Cousin(?X, ?Y)

Goal: {AmericanCousin(John, ?X1)}

{American(?Y), Cousin(John, ?Y)}

Find an American and check whether she or he is a cousin of john. And lets say your program has access to the entire database of the US of the entire world for that matter. We can see that what this is going to do is for every person which it can know to be American it will ask the subquery the second query that is that person a cousin of john. Now obviously this is a grossly inefficient way of doing things. Instead if we had written the code like this. What is this saying find a cousin of john assuming the same goal and test whether he or she is American.

(Refer Slide Time: 37:03)



Order matters ! (in Prolog)

Top to down , left to right

AmericanCousin(?X, ?Y) ← American(?Y), Cousin(?X, ?Y)

Goal : {AmericanCousin(John, ?X1)}

{American(?Y) , Cousin(John, ?Y)}

find an American and check whether s/he is a cousin of John !

AmericanCousin(?X, ?Y) ← Cousin(?X, ?Y), American(?Y)

find a cousin of John and test whether s/he is American.

And as you can see because you expect john to have very small number of cousins lets say whatever 5 10 15 20 something like that not in millions. So all you would do is you would test for this 15 20 people and see whether any of them is American. And if any of them happen to be American you would return yes so and so is American cousin of john. Other wise you would return no saying no john doesn't have any American cousins. Whereas the first program looked at all americans before saying john doesn't have an American cousin if none of them happen to the a cousin of john so obviously the first one is much much more inefficient than the second one. So all that we have done is that logically we have not changed the definition we have said that an American cousin means you must be American and you must be a cousin. But which of those two sentences you make first matters much more than prolog. So I will stop here and next class we will explore this idea of writing things in the correct order miuch more.