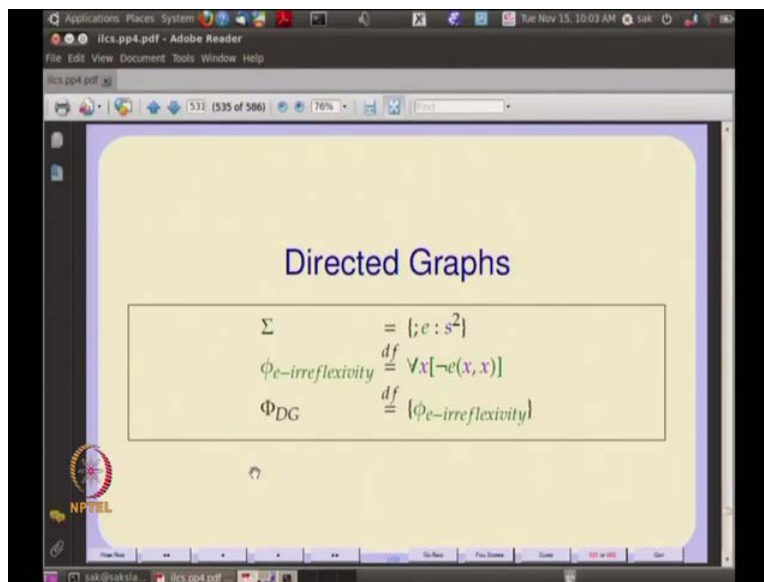


Logic for CS
Prof. Dr. S. Arun Kumar
Department of Computer Science
Indian Institute of Technology, Delhi

Lecture - 36
Towards logic Programming

Actually many of you have actually studied some amount of prolog and programming languages. But, it is a good idea to look at it from the point of view of look at logic programming which is different from prolog at least lightly different from prolog from the point of view of or it might be called first order theories.

(Refer Slide Time: 01:04)

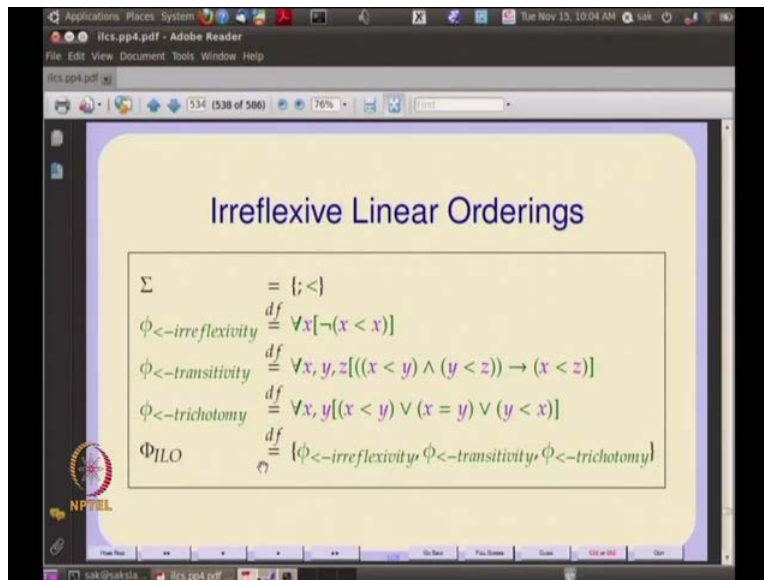


The screenshot shows a presentation slide titled "Directed Graphs" within an Adobe Reader window. The slide content is as follows:

$$\Sigma = \{e : s^2\}$$
$$\phi_{e\text{-irreflexivity}} \stackrel{df}{=} \forall x [-e(x, x)]$$
$$\Phi_{DG} \stackrel{df}{=} \{\phi_{e\text{-irreflexivity}}\}$$

The slide also features an NPTEL logo in the bottom left corner and a navigation bar at the bottom.

(Refer Slide Time: 01:16)



So, the last time we look at first order theories so, we look at for example this kinds of the notion of directed graphs. And, then undirected graphs and in Irreflexive partial orderings Irreflexive linear orderings. And, in all these cases what you are essentially saying is your first order theory is just all the logical consequences of the set of axioms. So, for example this set capital pi i a low which contains all these axioms for Irreflexive linear orderings. You, take all the logical consequences of these axioms. In, some sound and complete proof system for first order predicate calculus with equal usually equality. Then, all the logical all the statement that are logical consequences of these axioms and of first. So, that includes logical consequences include all the valid statements of first order predicate calculus involving these all the and all the statements that are derivable using lets a some proof system.

So, in particular one proof system on can use so it could be a tableau proof system or it could be a resolution proofs system or even it could be style proof system. But, the whole point about the style proof system is that or the natural reduction proof system is that, it is not very directed and, for it is not very deterministic. So, it is not clear how to automate the proof because most proofs by the style system require some impression of how one would go about trying to prove a logical consequence. On the other hand resolution in the case of tableau of course what you are saying is then you have to know the logical consequence. That, you are trying to prove take the negation of that and essentially obtain a close tableau. If, you can get a close tableau then of course you

are prove in it. In the case of resolution is similar to the tableau you take the negation of the logical consequence you are trying to prove. And, you try to do all possible try to find all possible resolvents. And, any anyone resolvent which leaves you to the empty clause is a correct. Which, leaves you to an empty to the generation of an empty clause is actually a correct proof of their logical consequence means that so that is the perspective in which we should look at. So, we are looking essentially first order theory. So, And why we are restricting as saying to first order theories? In general even if certain theories of higher order we could actually do some hackwork. So, one of the hackwork that you can do is to actually import the entire higher order carrier set. So, if you are looking at both higher order properties of lets a numbers. Let us say second order properties then essentially you are looking at properties of subset of the number of the set of naturals I say.

So, then what you could actually do is in your signature you could actually put in all those relations as part of the signature. Which, have to do with the second order functions a second order relations. And, in your carrier set for the models can include both the naturals and subsets of naturals. And in which case if your carrier set includes them then those second order properties or numbers. Since, they are part of the models itself they become first order properties of these larger carrier set which includes subsets of numbers 2. And, the reason reasoning mechanisms then are essentially the same there has to be some interface which some operations which allow you to extract things from sets of numbers treat them as individuals so and so on so. Which, means that you require also a tight system which distinguishes whether a certain individuals represents a number or set of numbers. So, the types of system in so in order to have the type system. What, you need to do is you need to add extra relations like is number is set of numbers and axiomatize them to. So, it is not possible it is not necessary that you can do all higher order that way. But, you can do a limited amount I mean you can just first order and second order. Let say properties of numbers and sets of numbers I mean that is you can make it all a part of the logic. The reasoning mechanisms will be essentially the same and in fact what happens a in logic programming is essentially that certain limited number of orders or part of the axiomatization that you actually put in your a logic programs so, that is the basic idea. Since, the reasoning mechanisms are essentially the same.

(Refer Slide Time: 07:06)

The slide is titled "(Reflexive) Partial Orderings". It defines a structure Σ as $\{; \leq : S^2\}$. It then lists three properties with their definitions:

- ϕ_{\leq} -reflexivity $\stackrel{df}{=} \forall x[x \leq x]$
- ϕ_{\leq} -transitivity $\stackrel{df}{=} \forall x, y, z[(x \leq y) \wedge (y \leq z) \rightarrow (x \leq z)]$
- ϕ_{\leq} -antisymmetry $\stackrel{df}{=} \forall x, y[(x \leq y) \wedge (y \leq x) \rightarrow x = y]$

Finally, it defines Φ_{PO} as the set of these three properties: $\{\phi_{\leq}$ -reflexivity, ϕ_{\leq} -transitivity, ϕ_{\leq} -antisymmetry $\}$. An NPTEL logo is visible in the bottom left corner.

We can look at logic programming essentially as a logic program as an essentially an axiomatization of the properties of some structure that you are looking at first order structure.

(Refer Slide Time: 07:25)

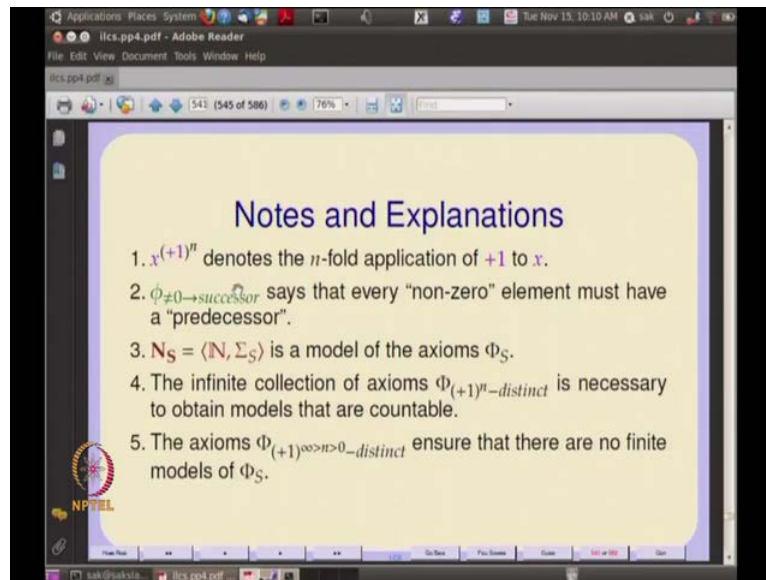
The slide is titled "Peano's Postulates" and lists five postulates:

- P1. 0 is a natural number.
- P2. If x is a natural number then x^{+1} (called the *successor* of x) is a natural number.
- P3. $0 \neq x^{+1}$ for any natural number x .
- P4. $x^{+1} = y^{+1}$ implies $x = y$
- P5. Let P be a property that may or may not hold of every natural number. If
Basis. 0 has the property P and
Induction Step. whenever a natural number x has the property P , x^{+1} also has the property P
then all natural numbers have the property P .

An NPTEL logo is visible in the bottom left corner.

That is what so, we had all these like reflexive linear orderings and equivalence relations we had Peano's Postulates one interesting thing of course. And, so again you are the postulate number five P5 is actually higher order for any order.

(Refer Slide Time: 07:39)



But, what we will generally have is Peano's an induction Postulate which what for whatever relations that you are limited to in your signature that's what it will be. So, one for each relation that you have in your signature is what you will get as P5. So, what you will get not the full power of Peano's Postulates. But, the restriction to whatever relations are defined in you are signature.

(Refer Slide Time: 08:07)

The slide is titled "Finite Models of Arithmetic" and is displayed in a window titled "ilcs.pp4.pdf - Adobe Reader". The slide content is as follows:

Finite Models of Arithmetic

1. If the infinite collection $\Phi_{(+1)^{\infty>n>0}\text{-distinct}}$ is replaced by a finite collection for some $m > 0$ i.e.
$$\Phi_{(+1)^{m>n>0}\text{-distinct}} = \{\phi_{(+1)^n\text{-distinct}} \mid m > n > 0\}$$
then both finite and countable models are possible.
2. If in addition to $\Phi_{(+1)^{m>n>0}\text{-distinct}}$ we also include the axiom
$$\psi_{\text{modulo } m} \stackrel{df}{=} \forall x [x^{(+1)^m} = x]$$
we get models $\mathbf{Z}_{S,m} = \langle \mathbf{Z}_m, \Sigma_S \rangle$ for the integers modulo m and there are no infinite models.

(Refer Slide Time: 08:12)

The slide is titled "A Non-standard Model of Arithmetic" and is displayed in a window titled "ilcs.pp4.pdf - Adobe Reader". The slide content is as follows:

A Non-standard Model of Arithmetic

Consider the model $\mathbf{N}_S = \langle \mathbf{N}, \Sigma_S \rangle$ of the axioms of **number theory**.

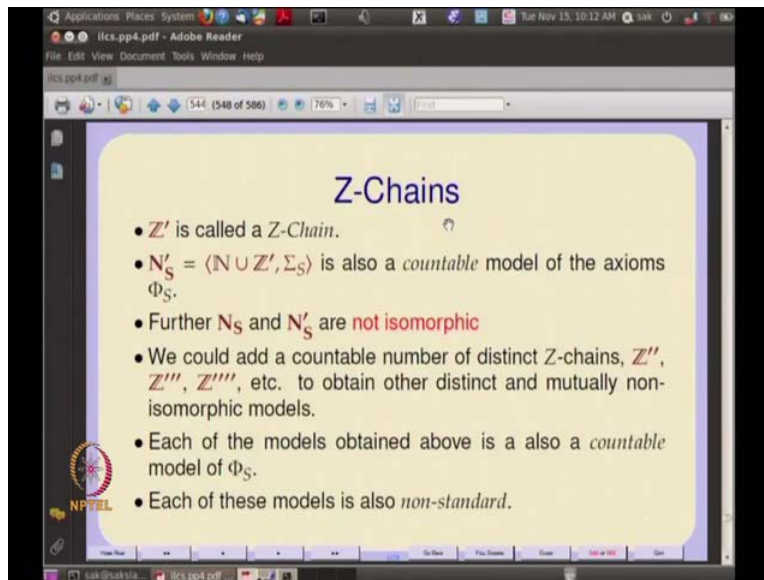
- We add a new element $0' \neq 0$.
- This implies adding an infinite number of new elements $0^{(+1)^n}$ one for each $n > 0$. For simplicity let us call these elements $1', 2', 3', \dots$. Each of these new elements is different from every element in \mathbf{N} .
- Since $0' \neq 0$, it must have a "predecessor" say $-1'$ which again leads to the addition of all the elements $-2', -3', -3', \dots$ each of which is distinct and different from all other elements. Let us call this set of elements \mathbf{Z}' .

$\mathbf{N}'_S = \langle \mathbf{N} \cup \mathbf{Z}', \Sigma_S \rangle$ is a model of Φ_S and is said to be *non-standard*.

So, well then we also look at Finite models and if you look at a Nonstandard Models. And, we essentially showed that you can have accountable number of nonstandard models of arithmetic. which but it is. So, the usual model of arithmetic consisting of just the naturals which is like the smallest set that can be generated with 0 and, successor operation. And, is closed under successor operation that, smallest set is called by standard model of arithmetic. So, implicitly in the standard model of arithmetic what you are essentially saying is that. You, are not considering

any other junk elements which may be added like this, 0 prime or 1 prime or 2 primes. You, are not adding any more junk elements so, you are looking at the smallest set. That is only implicit and it is not actually expressed as a first order logic formula.

(Refer Slide Time: 09:12)



The image shows a screenshot of a presentation slide titled "Z-Chains". The slide is displayed within a window titled "ilcs.pp4.pdf - Adobe Reader". The slide content is as follows:

Z-Chains

- \mathbb{Z}' is called a Z-Chain.
- $\mathbb{N}'_S = \langle \mathbb{N} \cup \mathbb{Z}', \Sigma_S \rangle$ is also a *countable* model of the axioms Φ_S .
- Further \mathbb{N}_S and \mathbb{N}'_S are **not isomorphic**
- We could add a countable number of distinct Z-chains, \mathbb{Z}'' , \mathbb{Z}''' , \mathbb{Z}'''' , etc. to obtain other distinct and mutually non-isomorphic models.
- Each of the models obtained above is a also a *countable* model of Φ_S .
- Each of these models is also *non-standard*.

The slide also features an NPTEL logo in the bottom left corner.

So, we had nonstandard models and we had what are known as Z- Chains basically you can create various images of the integers all distinct and not isomorphic each other. And, you can actually create countable number of them in nonstandard model. So, there are countable number of nonstandard models of arithmetic and, so those were some first order theories.

(Refer Slide Time: 09:42)

Reversing the Arrow

Let

$$\phi \leftarrow \psi \stackrel{df}{=} \psi \rightarrow \phi$$

Consider any clause $C = \{\pi_1, \dots, \pi_p\} \cup \{\neg v_1, \dots, \neg v_n\}$ where π_i , $1 \leq i \leq p$ are *positive literals* and $\neg v_j$, $1 \leq j \leq n$ are the *negative literals*. Then we have

$$\begin{aligned} C &\Leftrightarrow \forall [(\bigvee_{1 \leq i \leq p} \pi_i) \vee (\bigvee_{1 \leq j \leq n} \neg v_j)] \\ &\Leftrightarrow \forall [(\bigvee_{1 \leq i \leq p} \pi_i) \vee \neg(\bigwedge_{1 \leq j \leq n} v_j)] \\ &\Leftrightarrow \forall [(\bigwedge_{1 \leq j \leq n} v_j) \rightarrow (\bigvee_{1 \leq i \leq p} \pi_i)] \\ &\equiv \forall [(\bigvee_{1 \leq i \leq p} \pi_i) \leftarrow (\bigwedge_{1 \leq j \leq n} v_j)] \\ &\stackrel{df}{=} \pi_1, \dots, \pi_p \leftarrow v_1, \dots, v_n \end{aligned}$$

So, now let us towards logic programming the first thing of course is that Robins did was he reverse the arrow. So, let us you define phi left arrow psi as being just an alternative way of writing if psi then phi. And, it is usually red as pi if psi let us take any clause C. So, we are essentially looking at logic programming as through resolution. So, let us take any clause C I can partition that clause C into their positive and negative literals. So, let us assume that there are p positive literals and there n negative literals and giving them distinct names. But, there is negation symbol before each of them. Remember that in resolution during unification anyway the negation symbol is removed you are doing unification of complimentary pairs after removing the negation symbols.

So, now essentially this clause is logically equivalent to the universal closure of all the free variables that occur in this predicates. And, therefore it is a clause disjunction universal closure of the disjunction of all the literals both positive and negative literals. And, of course if you look at the negative literals it is an OR of NOT. And so Demorgan's law is applicable. Which, means that is logically equivalent to not of and. Where, I essentially factored out the negation symbol from the negative literals. And, when I have this naught then this is essentially like saying that this conjunction of the literals which occur in negatively take that positive forms take the conjunction of the positive forms of the negative literals. And, a that should essentially logically imply the disjunction of the positive literals. Which, by reversing the arrow essentially get that

the disjunction of the positive literal is true. If, the conjunction of the positive forms of that negative literal is true. So, this clause may therefore be written in this form will use this is the notation for the clause. So, that negation the negation symbol does not appear any. So, the to the left of the left arrow the commas indicate a disjunction or and to the of the left arrow the commas indicate conjunction. So, you are taking conjunction of new one to new n here you are taking disjunction of ϕ_1 to ϕ_p . And, you are saying that disjunction of ϕ_1 to ϕ_p is true if this conjunction of μ_1 to μ_n is true subject to an implicit universal closure of all the free variables.

(Refer Slide Time: 13:05)

Horn Clauses

Definition 36.1 Given a clause

$$C \stackrel{df}{=} \pi_1, \dots, \pi_p \leftarrow v_1, \dots, v_n$$

- Then C is a Horn clause if $0 \leq p \leq 1$.
- C is called a
 - program clause or rule clause if $p = 1$,
 - fact or unit clause if $p = 1$ and $n = 0$,
 - goal clause or query if $p = 0$,
- Each v_j is called a sub-goal of the goal clause.

So, what one can do now is one can consider a restriction of this notion of clause. In, which the set of part positive literals is restricted to being at most 1 a single term set of positive literals. And, the rest there essentially negative literals and we are never ever going to use the negation symbol. But, so this is restriction of the language I mean so, let us look at or first order logic in prospective. So, we had this full first order logic then we had prenex normal forms. Which is restriction of the language to those in which the quantifiers got factored out. And, of there is purely propositional body of predicates following the quantifiers. And, in the case of the prenex normal conversion you actually preserved logical equivalence. So, in that since by restricting the syntax of the language you have not restricted the meanings in the models in any way.

So, this smaller language of prenex normal forms is good enough for the whole of first order logic you don't require the full first order logic. Further of course in the case of prenex normal forms from proposition logic we know that we know that. If, you restrict proposition logic to just conjunctive normal forms then there is there also logically equivalent to just conjunctive normal form. So, a restriction of proposition logic to pure conjunctive normal forms does not in any way restrict the expression of the language. So, which gave us prenex conjunctive normal forms. So, every first order logic formula is logically equivalent to another first order logic formula in this restricted clause in this restricted language of prenex conjunctive normal forms.

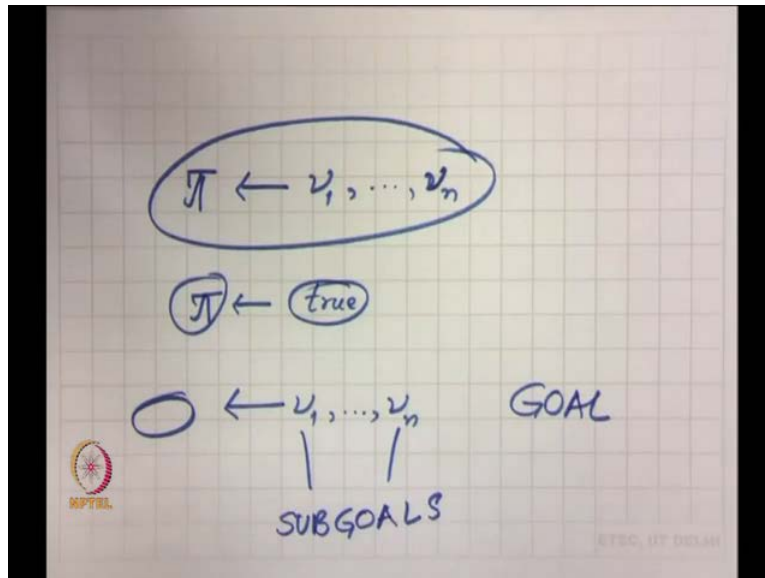
So, far logical equivalence was preserved and therefore expressiveness of the language was not was not in any way alter. Then, what we did was we sured a we colonized by removing by replacing existential quantifiers in the prenex conjunctive normal form by appropriate function symbols. By, adding expanding the signature in this process we actually lost logical proofs. So, thus colonized forms a no longer logically equivalent to the original formula because there are models as for as satisfy. So, in what we have lost logical equivalence in the sense that there are models of the original formula. In, the sense that if the original formula is satisfiable then, there is a Herbrand's models we satisfies this colonized form with the expanded signature. So, the expanded signature essentially gave you different classes of models under interrelationship between the models. So, all the models of the original formula were there in the expanded in this colonized form but, the colonized form actually use an, expanded signatures. So, in that sense as far as logical equivalences concerned the two sets of models are not really comparable.

So, what we preserved in the process of colonizing was the property of satisfiability or unsatisfiability. And, anyway by Herbrand's theorem we know that, satisfiability or unsatisfiability essentially is restricted to the Herbrand's universe it is sufficient to restricted to the Herbrand's universe. So, in colonization we did not preserve logical equivalence because anyway the signatures are different. And, therefore are not comparable however we preserved satisfiability the existence or non existence of a model. That, question could be equally well answered in the colonized form as in the original form. So, that was the first compromised with a logical equivalence. Now, a in the class so resolution therefore since it preserves the notion of satisfiability or unsatisfiability it was sufficient therefore to use colonized forms. And, let us use resolution in order to prove in order to refute in order to prove logical consequence. You, just

took the negation of the logical consequence and deduct resolution refutation. So, there that is where a, that is first place we are compromised on logical equivalence. And, now here is a second place that we compromising. So, if you are restrict your language to Horn Clauses. Now, you are not even you are not even preserving basic satisfiability or unsatisfiability. So, this is a restricted language the only thing about this language of Horn Clauses is that, the Clauses of Horn clauses is exactly corresponds to the clause of computable functions. And, the resolution a proof procedure that that is going to be used for horn clauses. Therefore, will do inductive computations very much like in function programming language. So, there is a certain things which I have been done like the undesirability of the validity problem. So, there are undesirability or validity of a formula is intrinsically not compute not necessarily computable in the sense. That, there is no general purpose algorithm which given any first order logic formula will be able to tell you whether it is valid or not. Where, as in the case of Horn Clauses a more or less you can think of it as sot of desirable subset proving validity for restricted version of the first order logic. But, the restriction is such that, the notion of computability can still be captured using horn clauses. So, in that sense the subset the language subset of horn clauses exactly corresponds to the computable function torturing computability.

So, whatever is decidable in Horn Clauses what is computable also. So, most computation can be converted into a decidability question so, that it gives an answer yes or no. Because, you can basically convert every function into a relation and then just looking for membership in the relation. So every decidability every computability question can be converted into a, decidability question and vice versa of given any decidability question you are just looking for Boolean function or Boolean computable function. So, that two way processes is start of and that, two way process is what you are looking at and these horn clauses exactly captured that. And, how to they so what happens so a clause C is a horn clause. If, there is at most one positive literal and all other literals are negative. So that is what $0 \leq p \leq 1$. Essentially says that there should there can be only 1 positive 1 literal on the left of the left arrow. So, this if p is equal to 1 then this is called program clause or rule clause. And it is a program clause or rule clause is a fact if there are no conditions you should read this as.

(Refer Slide Time: 22:12)



So, supposing you have supposing there is a positive literal ϕ then you are looking at so ϕ if μ_1 to μ_n . So, what you are essentially saying is a ϕ would be true if, all of these remember that negative literals you remove the negation and you take the conjunction of the literals. So, ϕ is true if μ_1 is true μ_2 is true and, dot μ_n is true. So, it is a conditional statement in that sense. And, if this n is 0 then you just have ϕ if nothing so that that what you are saying is that ϕ is true always that is called fact. So, this is equivalent to essentially ϕ is true if true, an MP because the conjunction of an empty set is true by the identity property. So, then this is this is of course logically equivalent to ϕ being true.

Therefore, ϕ is so this called a fact or unit clause if p is equal to 1 and n is equal to 0 and it is called a goal clause or query if you don't have anything on the left hand side. So, you just have some μ_1 to μ_n . And, you do not have anything here and this is this goal this notion and each of these so this is called a GOAL clause and each of these are called the SUBGOALS. So, you are asking essentially whether it is possible for μ_1 to μ_n to be all true simultaneously that is really what you are asking. So, what you are asking therefore is there are model of this conjunction of the set μ_1 to μ_n . Where, of course we are assuming that there all universally closed and so on so forth. I mean that it does not matter but you are just asking for whether there is a model. So, if you look at this goal clause.

(Refer Slide Time: 24:52)

The slide is titled "Goal clauses" and is displayed in a window titled "ilcs.pp4.pdf - Adobe Reader". The slide content is as follows:

Given a goal clause

$$G \stackrel{df}{=} \leftarrow v_1, \dots, v_n$$
$$\Leftrightarrow \vec{\forall} [\neg v_1 \vee \dots \vee \neg v_n]$$
$$\Leftrightarrow \neg \vec{\exists} [v_1 \wedge \dots \wedge v_n]$$

If $\vec{v} = FV(v_1 \wedge \dots \wedge v_n)$ then the goal is to prove that there exists an assignment to \vec{v} which makes $v_1 \wedge \dots \wedge v_n$ true.

The slide also features an NPTEL logo in the bottom left corner.

A, GOAL clauses form μ_1 to μ_n it is actually logically equivalent to universal closure of this disjunction of negations. Which, is equivalent to saying that there does not exist an instantiation of the variables which will make all of them true μ_1 to μ_n . So, the question if you look at it as query is there a model of which will make μ_1 to μ_n simultaneously true.

(Refer Slide Time: 25:38)

The slide is titled "Logic Programs" and is displayed in a window titled "ilcs.pp4.pdf - Adobe Reader". The slide content is as follows:

Definition 36.2 A logic program is a finite set of *Horn clauses*, i.e. it is a set of rules $P = \{h^1, \dots, h^k\}$, $k \geq 0$ with $h^l \equiv \pi^l \leftarrow v_1^l, \dots, v_{n_l}^l$, for $0 \leq l \leq k$. π^l is called the head of the rule and $v_1^l, \dots, v_{n_l}^l$ is the body of the rule.

Given a logic program P and a goal clause $G = \{v_1, \dots, v_n\}$ the basic idea is to show that

$P \cup \{G\}$ is unsatisfiable

$$\Leftrightarrow \vec{\forall} [\neg v_1 \vee \dots \vee \neg v_n] \text{ is a logical consequence of } P$$
$$\Leftrightarrow \vec{\exists} [v_1 \wedge \dots \wedge v_n] \text{ is a logical consequence of } P$$

The slide also features an NPTEL logo in the bottom left corner.

That, is equivalent to taking this adding into program clauses and proving that it is unsatisfiable. So, I have logic program so a, Logic Program is essentially set of horn clauses. And, given a set of horn clauses I give a goal clause then what I am asking is whether that whether it is true that given this set of horn clauses P this, is never possible. So, essentially that require that essentially means that if I include to goal clause in the set of program clauses I am asking whether the whole set is unsatisfiable. Now, if this whole set is unsatisfiable obviously so essentially what it means is that this existential closure of the conjunction is a logical consequence of the set of program clauses. Your, program clauses are essentially going to be a collection of essentially first order axiomatizations of some structure given some signature. So, let us look at logic programs from so here is a simple here is the there is a logic program is important this because it is allows you to do the axiomatization in top down fashion we cannot do arbitrary axiomatizations. Because, if it has to be executable then it has to be in some sense inductive I mean cannot be you cannot I, mean you still going to be still going to be bock down by induction. You cannot I mean you cannot I just like you cannot write functional programming like this.

(Refer Slide Time: 27:49)

$x \in \mathbb{N}$

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ f(x+1) \text{ div } (x+1) & \text{if } x > 0 \end{cases}$$
 ~~$f(x)$~~

$$\begin{aligned} f(0, 1) &\leftarrow \\ f(x+1, y) &\leftarrow \end{aligned}$$

NPTL
ETSC, IIT DELHI

Lets a factorial of x for natural x is a natural is 1 if x is equals 0 is f of x plus 1 divided by x plus 1 if x is greater than 0. No, I mean it is not suppose to be funny the whole point is that, this is non inductive definition of a perfectly correct valid axiomatization of factorial. This, is a perfectly valid axiomatization of the factorial function on the naturals. However, it is non

inductive and as a result this is not programmable. Now, you take you I mean there is absolutely nothing that prevents you from you taking a functional definition like this and, converting it into relational definition. And, if you convert it into relational definition then you will essentially. So I will essentially have two facts one is f of 0 comma 1 this is one fact. And, then the other fact I will have is f of x plus 1 y well if y equals if something it will have to be return it some in some complicated way. But, the point is it can be any functional definition can also be converted into relational definition. But, because of the fact that it is not inductive this logic program will never give any where as the usual definition of factorial is actually inductive.

(Refer Slide Time: 30:01)

$$x \in \mathbb{N}$$

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ \cancel{f(x+1) / (x+1)} & \text{if } x > 0 \\ x * f(x-1) & \end{cases}$$

~~f(0)~~ $f(0, 1) \leftarrow$
 $f(x+1, y) \leftarrow$

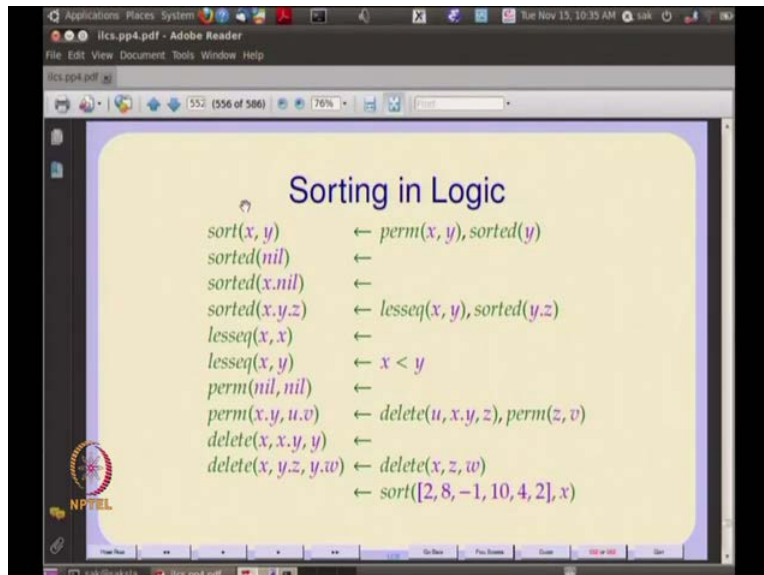
NPTEL ETSC, IIT DELHI

Which, makes this f of x minus 1 multiplied by x which converted relationally becomes inductive also. And, therefore it is a program because there is a well ordering guaranties termination the original definition is not guaranty to terminate that is the basic point. So, your logic programming for and axiomatizations are therefore limited by induction by well orderings you have to be able to show that, there exists a measure positive non negative measure. Which, is guaranty to decrease with every recursive call and it has to be always non negative. So, therefore it has to terminate it is bounded below by 0 that is it.

So, you have to show those things so in fact that is true of all your programs in language also when you take your while loops you have to have there is a measure. Which, you may or may

not have realize if that which guaranties termination of the while loop if does terminate. If that while loop does not terminate then there is no non negative measure which decreases or at least there is a at least some instance. Where, the measure may not decrease even if It is not negative there are some iterations where it will not decrease. So, that is in fact the basic idea of in vary look in variants in while programs and there the notion of termination recursion termination so and so forth. That, there has to be a measure which I can specify and finite well ordering set which ensures that I cannot I will never get out of that set. And, which ensures that which each recursive call or with each iteration I step down at least once at least one step. And, that would mean in the proof of correctness of your of termination of your programs.

(Refer Slide Time: 32:26)



So, let us look at Sorting is a so what we are going to do is, so, what is signature here. The signature logic program is not as sophisticated as type functional languages. So, it does not actually have a type system the type system is therefore the responsibility of the user so you have to program all the types as predicates if you want. But, broadly speaking in this program you can think of you can assume that the basic hardware is available the signature corresponding to your basic hardware is available. So, numbers are available list formation is available cons the cons and nil operations are available on lists. So, your signature consists of let us say the integers all integer arithmetic operations the most basic integer relational operations relations basic list constructors. The cons constructor and the nil constructor which allows you to construct lists of

numbers and basic equality of course. So, then you are looking at a sorting your restricting yourself to that subset of the term algebra. Which, is obtained only from a nil and consing of numbers to integer lists.

So, you are restricting yourself to that signature and you are restrict and you are using the relation the relational operations on the. Since, you are using numbers anyway you are using the relational operators relational the relations predefined or numbers. And, now what you are doing is you are trying to give a first order axiomatization of sorting. So, as far as sorting is concerned of course we have to we cannot think of 1 number being sorted we, have to think of lists. But, that is something we have to specified some point.

So, here for the movement assume that x and y are lists, then this essentially says that y let say y is the sorted form of x . If, y is a permutation of x and y is sorted at the movement of course. Since, we are not specified any these x and y could be anything. The other thing of course is standardizing of variables a part means that this x and this y refer to this x and this y refer to this y . But, any other x is in the rest of the program are different from those from those x . So, you can call this $x_1 y_1$ it be like so in fact any prolog system will first do that it will first standardize a variables apart by renaming them by generating temporary variables. And, creating a table of associations between the generated temporaries and the program use a program or names. So, this essentially what we are saying is we would have axiomatized the notion of sort sorting provided you can axiomatized the notion of the permutation.

And, you can axiomatize the notion of sortedness what does it mean for something to be sorted. The, nil constructed implicitly is that of lists and what you are saying is that an empty lists are already sorted so, this is the fact. So, you can see that there is no condition here so this is fact. The empty list is always sorted the one element list so this dot stands for the cons operation on lists. Therefore, implicitly this x has type which is often individual not of a list. Whatever, it is whatever may be the type you might assign to this nil. Even, if this nil refers to the empty list in list of lists then, this x refers to a list an individual in that list of lists and an individual in list of lists is a list is simple list. So, what at this movement what you are saying is that any 1 element list is sorted. And, then you have an inductive call which says supposing you have list containing at least 2 elements. So, this the cons of course is associated. So, this is so assumes that it is a list y cons z then x cons y cons z . So, this is sorted provided in that in the space of individual terms I

mean a, the color coding here is deliberate the violet is because of I am really looking terms in that signature.

And, the green is because I am looking a predicates. This, there is a property of less than or equal to which should be which should hold of x and y . If, that holds and if y cons z is sorted this is the inductive definition. Then I would claim that x cons y cons z is sorted z would implicitly be a list. Notice that, sorted is recursively defined here or rather actually inductively defined. So, z would have to be a list it cannot be any division because you will have to consider possibility of z being nil that will automatically type it to be a list. And, for the possibility of z being some x dot nil or x dot some z prime which will type them all to be appropriate I mean. So, you are looking at certain constructors only like so of course. So, the axiomatization of sorted so sorted has 3 axioms of which true or unconditional. And, the inductive 1 is conditional and it is based on axiomating less or equal to and that is what depends upon. So, now the natural thing is to axiomatize less than or equal to. So, of course less than or equal to for any of this is individuals because look at the second clause x is less than or equal to y . If, x is less than y where this less than is part of the signature on the individuals let say the numbers. So, then every element is less than or equal to it itself every individual. So, these two axiomatize less than or equal to completely. And, given that these two axiomatize less than or equal to you have axiomatized sorted. And, now in order to axiomatize sort you need to axiomatize the notion of permutation. Basically, you list the sorted only if there is a permutation of that list which is ordered which is sorted. So, the notion of permutation is very simple the empty list is it is not permutation and there is no other permutation. Now, if I have two lists x dot y u dot v so x cons y is a permutation of u dot v or other way let say u dot v is a permutation x dot x cons y . If I can if from x dot y I can some out delete you and I get a smaller list let us call that z . And, this v is just permutation of that z so here again we are looking at inductive.

So, this v has to be a permute if v is a permutation of z and, in x dot y basically if I put back you in in z if i put back you at an appropriately if I can get x dot y . Then, I would claim that x dot y is essentially permutation of u dot v . So, this permutation so, the axiomatization of permutation realize on the axiomatization of the delete. And, here you have to realize at this delete has to be an explicit delete. So, it assumes the presence and deletes it does not ignore if u is not present in the list. It, actually gives you an answer of no if you cannot be found in the list. So, this delete is

an explicit delete and it is essential to have an explicit delete in order to retain the inductive nature of our definition. Notice that if, you were not present and you gave the answer yes then you are not stepping down the ordering in a well order. I mean the length of the list is not changed is non decreasing with each recursive call. The length of the list you are considering in each recursive call is not decreasing and, therefore your termination is not guaranteed. So, here is where I mean so all these, programming equation is hidden in the axiomatization. So, delete well is again an inductively defined thing. If I, have the list $x \cdot y$ and I am deleting x where x is the first element then I just get back y .

So, the effect of deleting the first element gives me the rest of the list so, that is a fact. Now, if I am want to delete x from a list called $y \cdot z$ and let us assume x is not the same element as that of y . Then, what does it mean I had to inductively go down z so, if I can delete x from z to give me a list w . And if I can put back that first element from $y \cdot w$ then, I can claim to have explicitly deleted x from $y \cdot z$. And, having deleted x from $y \cdot z$ I get $y \cdot w$. So, this set of and by the way so, this is of course this delete again is a inductive and up to here is the set of program clauses P . In, that sense this is the set that constitutes the logic program per say and what you would like to do is to give a something called goal clause like this. So, basically what you are doing is this goal clause actually automates a proof and actually determinizes and gives direction to the proof.

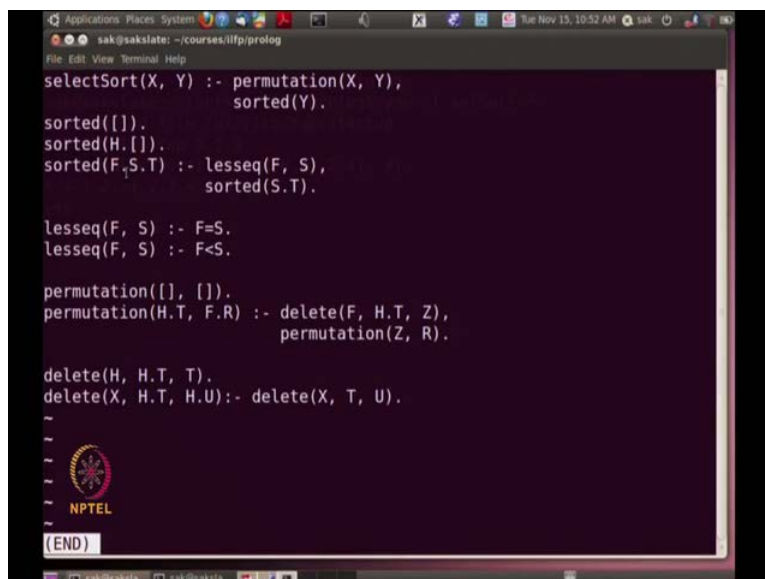
So, just think of it you are going to do resolution starting with this goal clause. When, you do resolution starting with this goal clause. What, you are going to do is remember that it is this is appearing on the right side of the arrow. So, therefore it is actually an negative literal a resolution will look for a positive occurrence of sort and then you try to unify them unify the parameters.

The only positive occurrence of sort is in fact here. So, when you take this set p and this goal clause p the effect of doing a finding a resolvent is to unify this x with this set containing $2,8$ minus $1,10,4,2$ and that automatically labels x as list let say. So, and it unify so, I have this variable x . So, this is this is of course let say I do not know $1,2,3,4,5,6,7,8,9,10,11$ let us call this x_{11} . So x_{11} is the same as y_1 I mean y_1 will so x_{11} will be substituted for y_1 as part of the resolution. So, a substitution is created in which x_1 is replaced by this set by this list. And, y_1 is replaced by x_{11} this substitution on the right hand side essentially therefore gives you that this is true. This, dedicates sort of this list $2,8$ minus $1,10,4,2$ comma x_{11} can be true. If, perm of this

list 2,8 minus 1,10,4,2 comma x 11. Because, y is replaced by x 11 first of all so if x 11 is a permutation of this. And, x 11 is sorted that is when it would be true. So, this permutation this list creating a permutation essentially means trying out all possible permutations of this list. Till you get some particular value of some particular permutation which is for which make this perm x,y true. Once, you got that permutation put it in this sort I mean you have created a fresh substitution for while for this y 1.

And, this y 1 has been substituted by x 11so, got a fresh substitution x 11 which is that permutation. And, that will be verified against the short list is it clear essentially you are going to generate all possible permutations. But, you are going to generate those possible permutations of this list directed by this list. So, you are only going to restrict yourself to the space of n factorial permutations of this list considering there are some duplicate elements they are slightly less than n factorial where, n is the length of the list. But, you essentially going to try out all possible permutations in some particular order till you get a permutation which is sorted. And, when you try out all possible permutations you are going to check for them being sorted through this. So, you keep creating fresh substitutions some of those permutations will fail at some point. Therefore, you will backtrack some previous substitutions to find out where and try out the next permutations and so on and so forth till you get a sorted form. And, this in fact what happens so this is logic programming I have actually got a programming log version here.

(Refer Slide Time: 49:34)



```
sak@saklate: ~/courses/ilffpip/prolog
File Edit View Terminal Help
selectSort(X, Y) :- permutation(X, Y),
                  sorted(Y).

sorted([]).
sorted(H, []).
sorted(F, S, T) :- lesseq(F, S),
                  sorted(S, T).

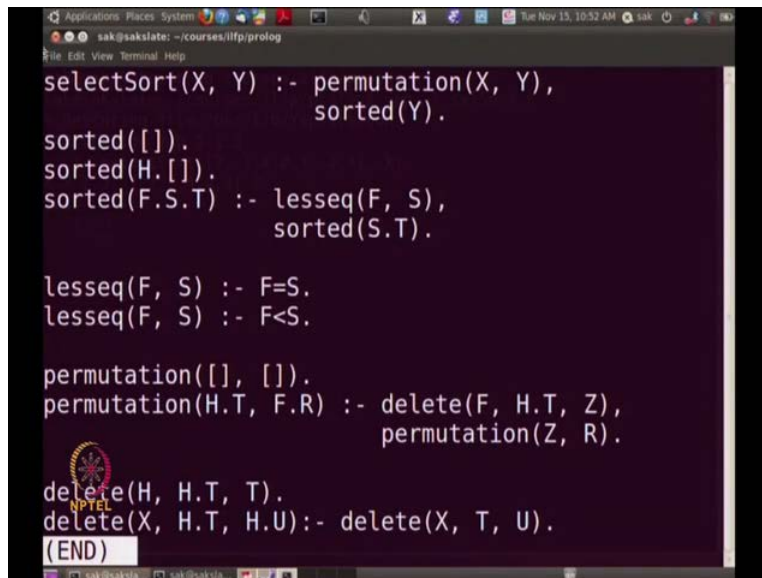
lesseq(F, S) :- F=S.
lesseq(F, S) :- F<S.

permutation([], []).
permutation(H, T, F, R) :- delete(F, H, T, Z),
                          permutation(Z, R).

delete(H, H, T, T).
delete(X, H, T, H, U) :- delete(X, T, U).

~
~
NPTEL
(END)
```

(Refer Slide Time: 49:56)



```
Applications Places System sak@sakslate: ~/courses/llfp/prolog
File Edit View Terminal Help
selectSort(X, Y) :- permutation(X, Y),
                    sorted(Y).

sorted([]).
sorted(H. []).
sorted(F.S.T) :- lesseq(F, S),
                 sorted(S.T).

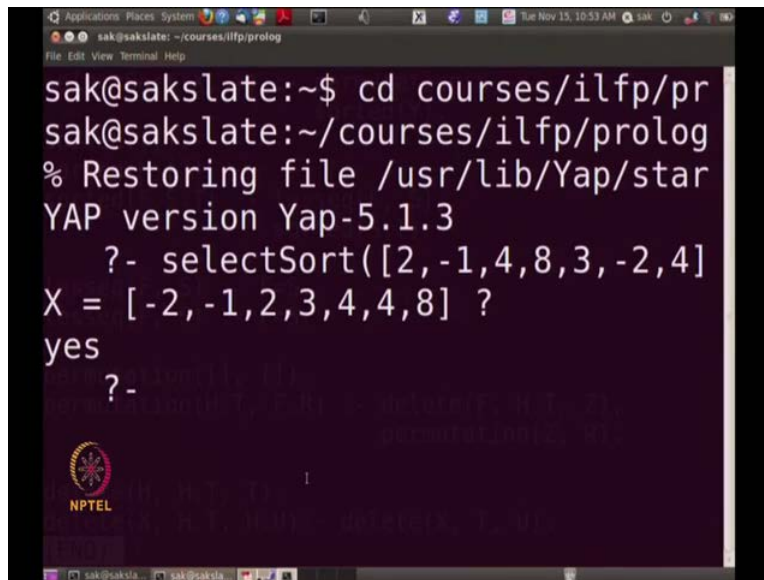
lesseq(F, S) :- F=S.
lesseq(F, S) :- F<S.

permutation([], []).
permutation(H.T, F.R) :- delete(F, H.T, Z),
                        permutation(Z, R).

delete(H, H.T, T).
delete(X, H.T, H.U) :- delete(X, T, U).
(END)
```

So, This is a selection sort procedure and prologue hope you can see it I do not think I can make it even larger than this is it I can just about see it there. Let us see what happens if I huge so the basic sorting axiom remains the same. But, what happens is since you are guided by inductive definitions. So, this is this remains more or less the same sorted less than or equal to. Here I have changed this to a different variables because prologue wants to be different F equals S of course. Since, we are dealing with numbers essentially equality defined. So, you are looking at first order theory is equality. And, then the less than is of course defined on numbers permutation here again prologue requires you to use completely distinct variables on all sites on the left side. And, then it will do the unification. So, you get this delete this and delete is defined by this.

(Refer Slide Time: 50:58)

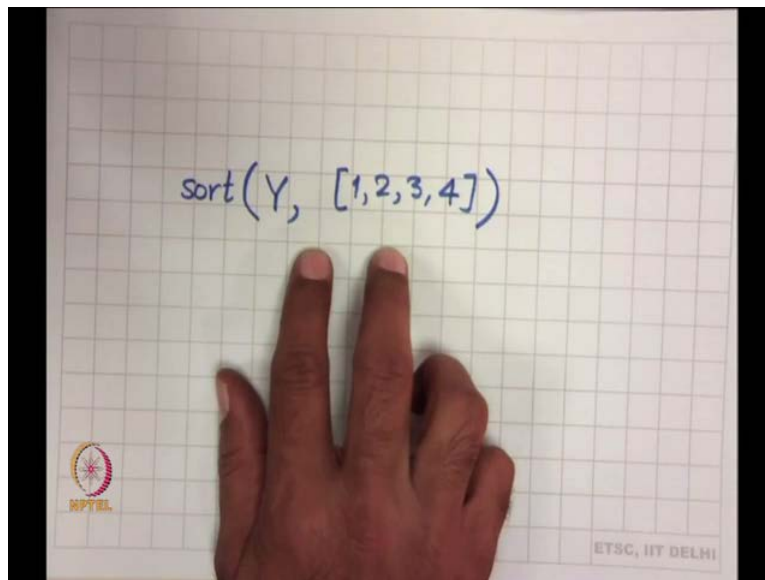


```
sak@sakslate:~$ cd courses/ilfp/pr
sak@sakslate:~/courses/ilfp/prolog
% Restoring file /usr/lib/Yap/star
YAP version Yap-5.1.3
?- selectSort([2,-1,4,8,3,-2,4]
X = [-2,-1,2,3,4,4,8] ?
yes
?-
```

Here, is an execution I am using something called Yap is probably the latest incarnation of prolog. And, is probably the fastest prolog engine that currently exists there are things like x is b and so on so forth. Which, used previously but they have quite of you bugs in them and pretty slow. But, this Yap is very powerful prolog system it is available on I do not know whether it is available on windows which definitely on Linux. So, you can take this and this is your query essentially this is your GOAL clause that mean what happened plus minus.

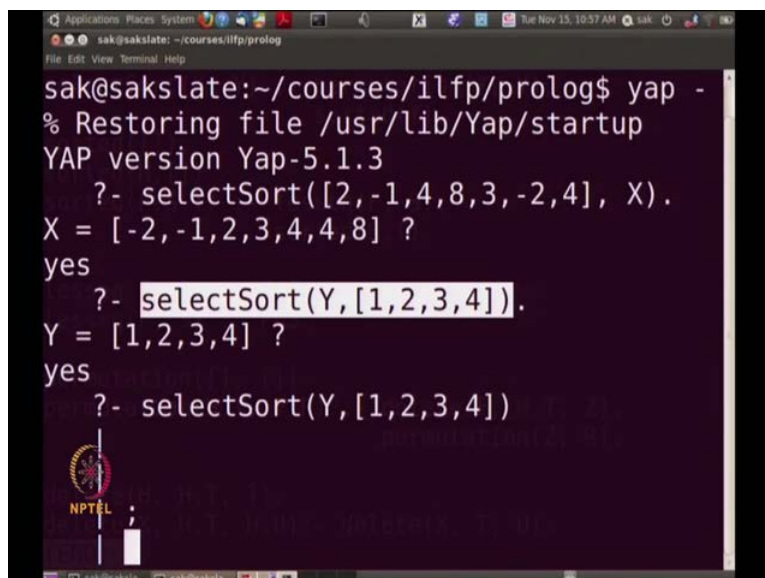
So, here is this so select sort call this is GOAL clause I am calling at with some variable x. And, it actually gives me this and gives me this yes keeps giving yes or no answer basically. So, there is there is of course one interesting thing about, this the notion of using relations is that unlike imperative programming languages or functional programming languages. There, is no notion of input or output it is axiomatizing a relation.

(Refer Slide Time: 52:59)



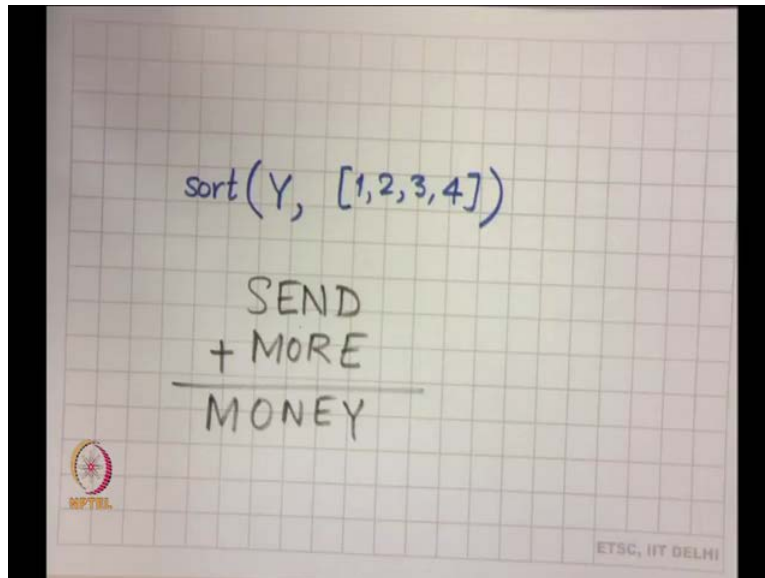
Theoretically speaking, it should be possible to give a query like this, sort let say Y and let say 1,2,3,4 there is there is no notion of input or output. There, no notion of an argument and result so that a symmetry has gone. Because, just doing first order axiomatization of relations actually it is an interesting thing. But, the problem is that this again is not inductively guided. But, theoretically speaking you should actually be able to get all possible values all possible permutations of this list as a result.

(Refer Slide Time: 53:09)



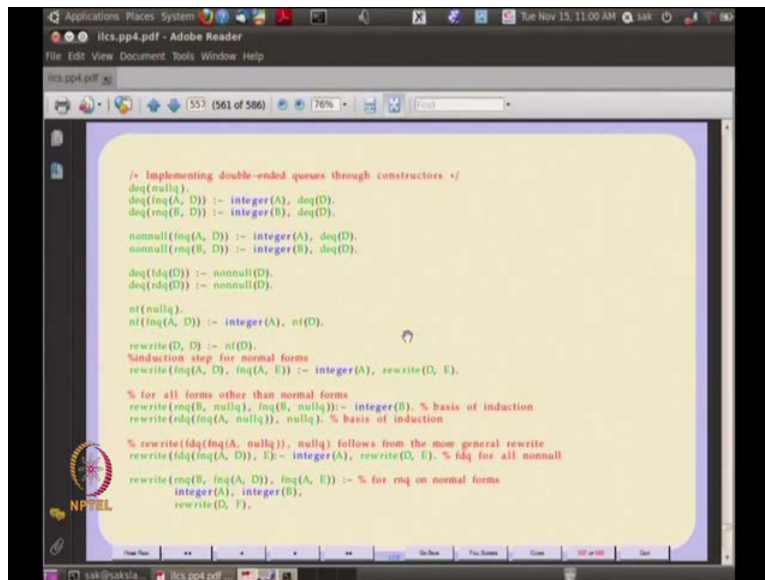
So, here is selection sort and there are various things but more various interesting things are a prologue is essentially for solving constraints. So, you want you want you want to essentially this is standard school type problem.

(Refer Slide Time: 55:50)



You, find digit is for all the letters such that this addition summation this addition some works. This is like, one of the standard puzzle problems which keep you occupied for whole day. So, here is a simple prologue program where there is this interesting thing here this equal to colon equal to, Have you ever counted this? Which, what this does is it takes the left hand side and the right hand side for all possible. So, we have before this we have made it clear that we just want to assign digit is to this list of letters. Which constitutes and MORE MONEY we have already restricted it to these 10 digit. And, we are doing varies assignments of digit is and what this whole thing does is it just constrains so, that this equation is true. So, it does a, unification within an equation theory basically. So, prologue is very good at that exhaust is such with backtracking to get all possible solutions.

(Refer Slide Time: 57:31)



```
/* Implementing double-ended queues through constructors */
deq(nullq).
deq(inq(A, D)) :- integer(A), deq(D).
deq(rmq(B, D)) :- integer(B), deq(D).

nonnull(inq(A, D)) :- integer(A), deq(D).
nonnull(rmq(B, D)) :- integer(B), deq(D).

deq(fdq(D)) :- nonnull(D).
deq(edq(D)) :- nonnull(D).

n(nullq).
n(inq(A, D)) :- integer(A), n(D).

rewrite(D, D) :- n(D).
%induction step for normal forms
rewrite(inq(A, D), inq(A, E)) :- integer(A), rewrite(D, E).

% for all forms other than normal forms
rewrite(rmq(B, nullq), rmq(B, nullq)) :- integer(B). % basis of induction
rewrite(edq(inq(A, nullq)), nullq). % basis of induction

% rewrite(fdq(inq(A, nullq)), nullq) follows from the more general rewrite
rewrite(fdq(inq(A, D)), E) :- integer(A), rewrite(D, E). % fdq for all nonnull

rewrite(rmq(B, inq(A, D)), inq(A, E)) :- % for rmq on normal forms
integer(A), integer(B),
rewrite(D, E).
```

So, you can this is something you can do and of course there is a, or there are various other things prolog has constructed. Which is called functors this is an implementation of double needed q's and i have I chose a data structure which is not often studied. But, you can read yourself so you can use constructors axiomatize those constructors and, play with prolog. But, the most important thing of course is that, there is a always there is a tuning machine implementation of prolog. Which I did not include maybe I mean since so there you can also program the tuning machine. Which, essentially is shows that prolog is as powerful as it can do all the tuning computer build relations. So, everything that I actually decidable can be programmed and prolog.