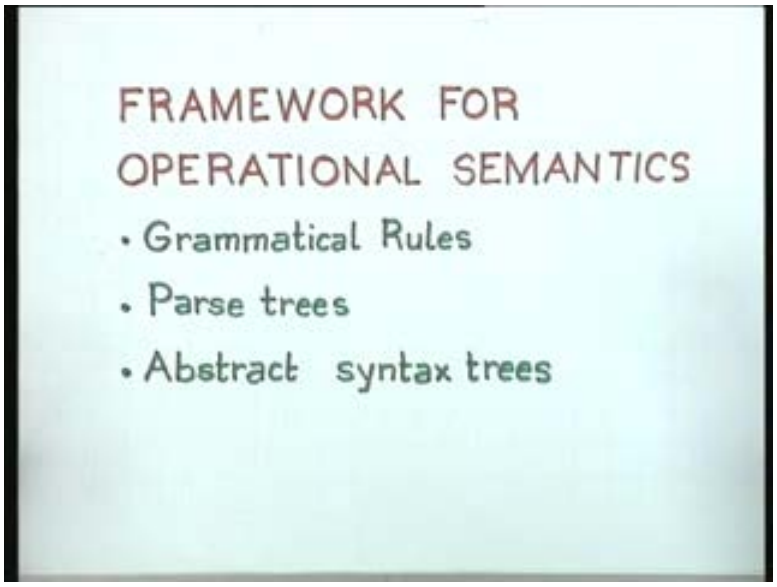
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture no 8
Lecture Title: Transition Systems

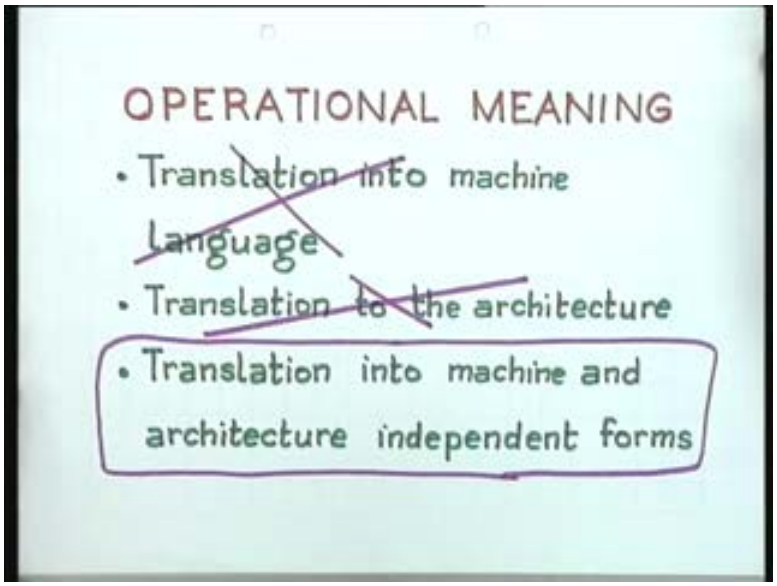
Welcome to lecture 8. I will just briefly go through whatever we did in the last two lectures and give a basis for what is known as transition systems for specifying the semantics of the language. Let us just briefly recapitulate the framework for operational semantics.

[Refer Slide Time: 00:52]



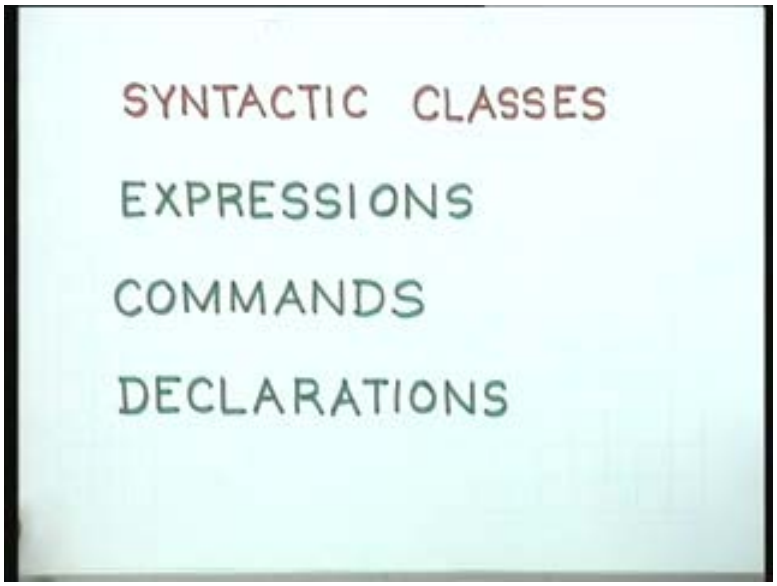
The basic structure on which our operational semantics is to be built is the grammatical rules of the parse trees but in a more abstract form than is really necessary for compiling. That way it is simpler. It has less number of grammatical categories which are important for meaning and so we will take usually some form of abstract syntax trees or their representation as a context-free grammar. While we are talking about semantics our context-free grammar could be ambiguous but that does not matter because we are interested in only the trees where there is no ambiguity.

[Refer Slide Time: 01:47]



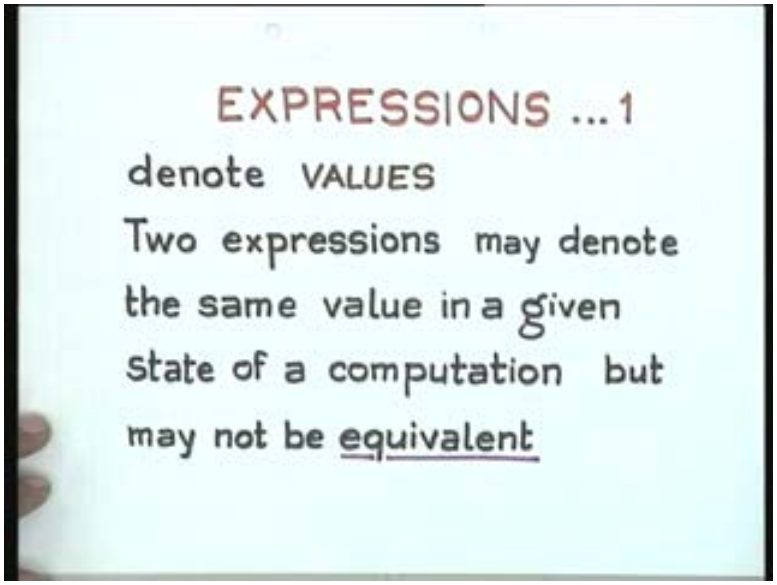
Let us go through operational meaning. Our main goal is to translate or to give a meaning of a language in some form which is machine and architecture-independent and it should not be to translate it into particular machine languages or into particular architectures. It used to be the case several years ago when the meanings of the constructs of many languages were defined in terms of some abstract machine. So, machine architecture was specified and everything was specified in terms of the running of this abstract machine but these days we would prefer to look upon the language as a separate object independent of any particular machine or architecture.

[Refer Slide Time: 02:45]

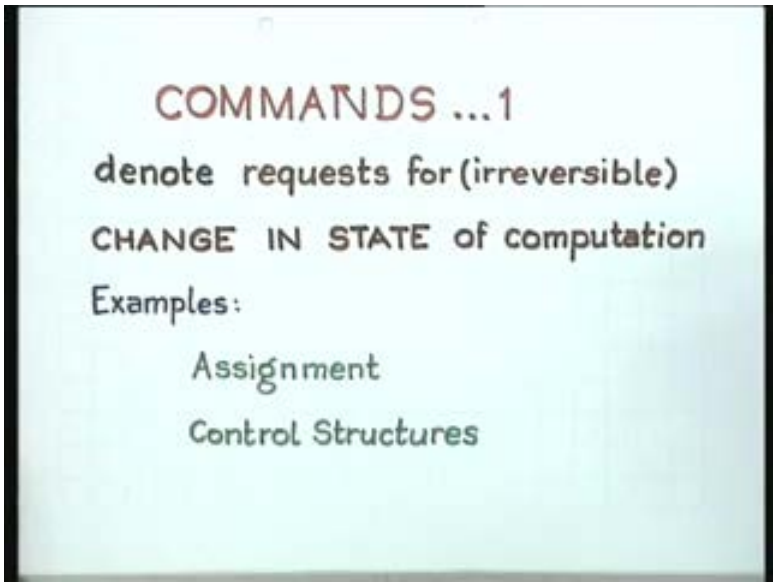


We would prefer to give it something that is independent of these particularities. We went through the basic syntactic classes of a language which we said were expressions, commands and declarations. These are syntactic classes which have specific meanings. Expressions just denote values. There is really nothing more to expressions. Whenever we talk about an expression its meaning is just a value evaluated in some suitable computational state. A computational state consists of an environment which gives the significance of the names used in the computation and also may be the notion of a store which stores values.

[Refer Slide Time: 03:10]



[Refer Slide Time: 03:45]

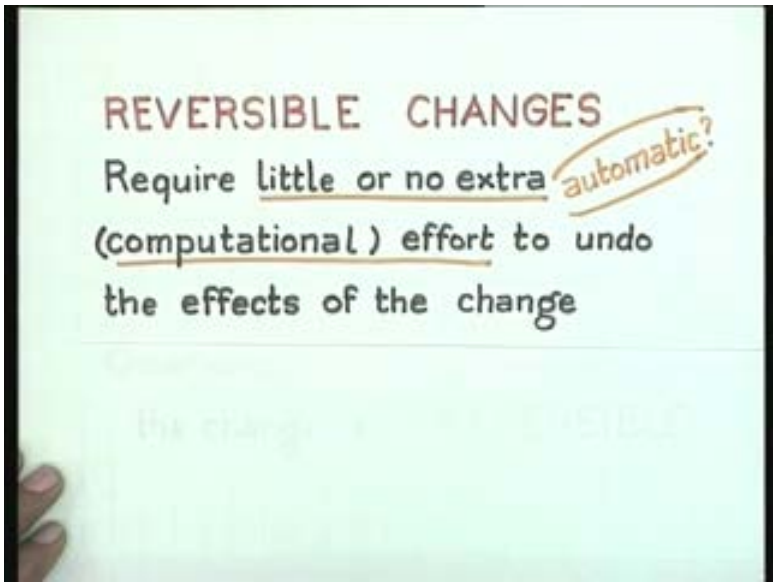


Commands, as I said, were denoted requests for irreversible changes in the state of computation which could mean changes either in the environment or in the store. Environments were created or changed by new declarations and declarations are reversible and the changes in the environment are reversible. Very often, declarations can also change the store and let us just go back for a clarification on reversible and irreversible changes. The analogy is really with thermodynamics. A change is irreversible not if it is impossible to undo the change but if undoing the change requires a great deal of work or energy to be expended whereas the change is reversible if it requires little or no effort to undo it.

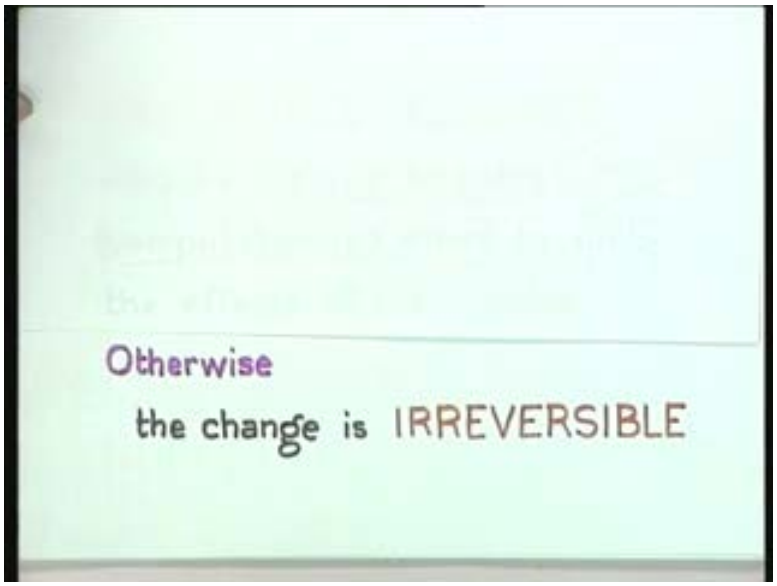
[Refer Slide Time: 04:03]

We will look upon these reversible and irreversible changes in essentially the same fashion. Very often a reversible change is one which can be undone automatically or with very little computational effort and an irreversible change is one in which you might have to expend at least as much computational effort to undo the change as you required to make the change in the first place. We call such changes irreversible.

[Refer Slide Time: 05:00]

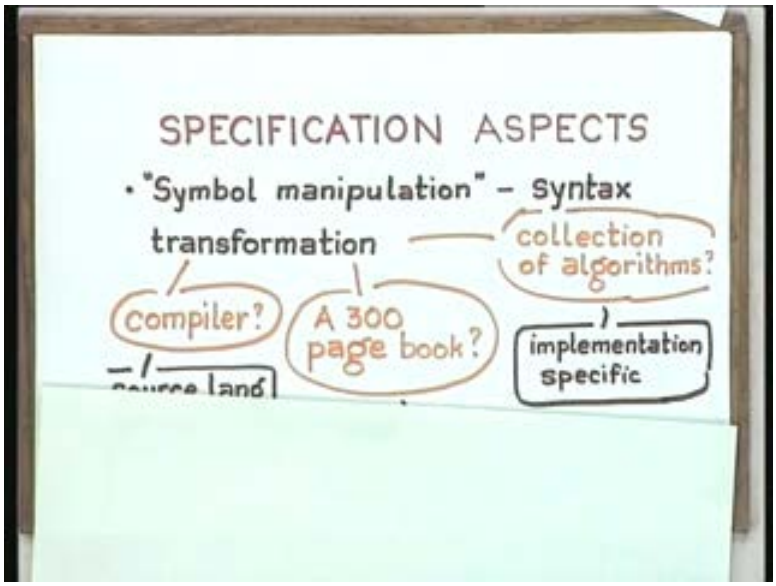


[Refer Slide Time: 05:18]



Let us study our specification mechanism. We are dealing mainly with syntax with no ‘a priori’ notion of meaning. At the background we should also remember that there are pragmatic aspects such as we should be able to implement these at the background though that is not most important. We will define a specification mechanism for specifying the meaning which is essentially syntactic in nature.

[Refer Slide Time: 08:36]



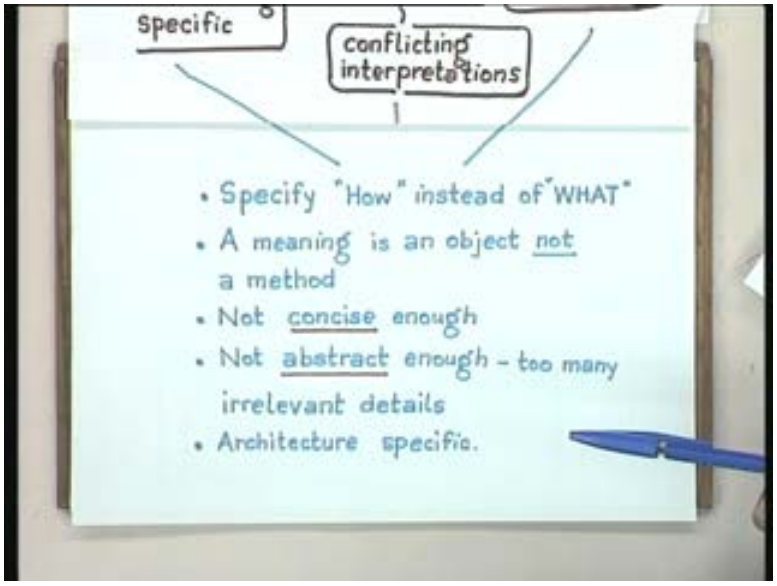
It is syntactic in the sense that it is mostly a form of symbol manipulation, symbol pushing, symbol deletion etc where these symbols belong to the abstract syntax tree of the language. Your semantics has to be firmly based on the syntax. The best way to give a meaning is in terms of the manipulation of symbols. After all that is really all that a compiler or an interpreter does. There are some languages whose first versions specified that the meaning of this language is exactly the compiler designed by us. But there are obvious pitfalls to it. First of all, it is terribly source-language dependent, terribly implementation dependent and if the language was not implemented in some bootstrapped fashion it would also be extremely machine dependent.

The other possibility is to just give a collection of algorithms for the various constructs of the language but even that is too implementation-specific. It is too pragmatic for example; no collection of algorithms is complete without a collection of data structures which are manipulated. This means that you are really designing an abstract compiler or an abstract interpreter which is again terribly pragmatic and it is really of no use to most people who are going to be users rather than implementers of the language. So, what is being done currently is to write a volume explaining the meaning of the language.

Most languages really have a huge reference manual which explains the various constructs in pitiless detail but they try to be abstract in the sense that they are not algorithmic, they are not machine-specific and they are expressed in some natural language which also has its own pitfalls. If you look at these three methods that have been used and are currently being used then what you find is that anything that is extremely pragmatic like a compiler or an interpreter or a set of algorithms has a danger. Firstly, it does not give the meaning of the language because it specifies how the language is to be implemented rather than what a construct means in the language.

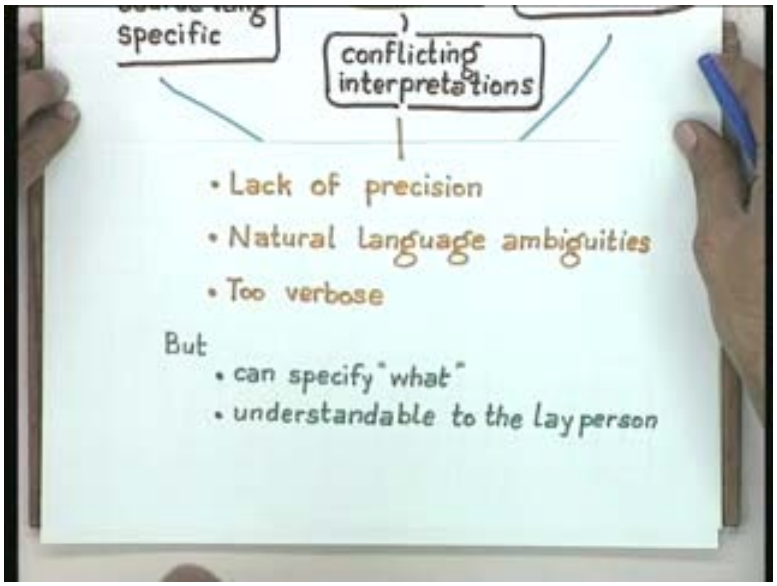
Secondly, when we look upon meaning of a language we regard that as an abstract mathematical object in itself. It is not necessary for it to be related to any particular method of implementation or any particular method of reasoning.

[Refer Slide Time: 10:12]



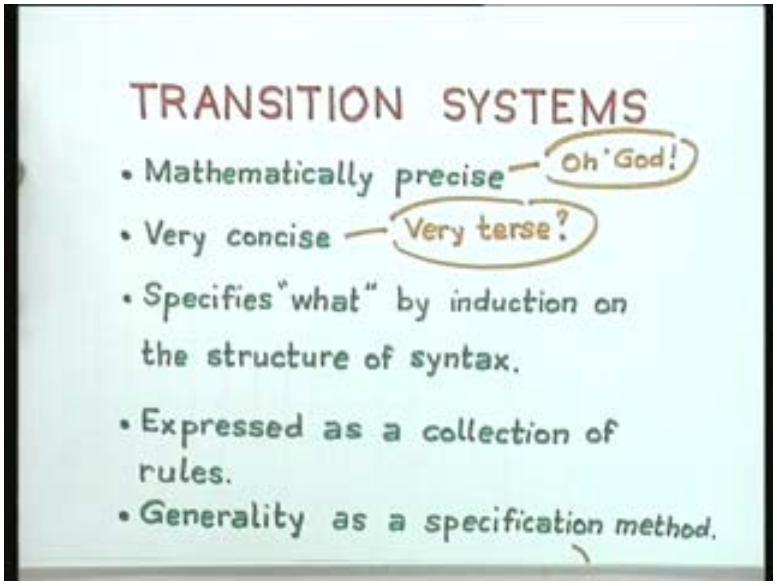
Even for a small language for PL0 the compiler runs into approximately ten pages. If the compiler were more user friendly it would probably run to more than twenty pages of actual source language code which makes it not too concise and because the PL0 language is a very simple language it is not really necessary to use the full detail of a compiler to explain the language. Lastly, they are not abstract enough. There are too many details which are irrelevant to the user. It is too architecture-specific in that even if it specifies an abstract machine with an interpreter for it, it is still architecture-specific and the main problem with this 300 page volume is that natural language by its very nature was not designed but evolved with shades and nuances of meaning.

[Refer Slide Time: 11:20]



Very often these shades and nuances of meaning are geographically specific and natural language is inherently ambiguous and too verbose. But the advantage it has is that it does convey a lot to a lay person and you can use natural language to specify what a construct means rather than how the construct is to be implemented or how the language is to be implemented. So, writing a manual for the language in some natural language is inevitable. That has to be there for a lay user but that still does not specify the meaning of a language in an accurate fashion. We will use what is known as transition systems.

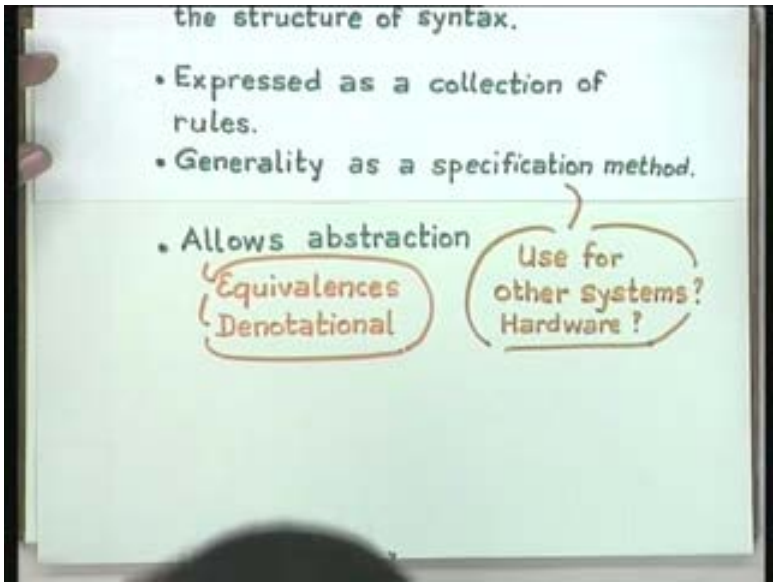
[Refer Slide Time: 12:45]



Transition systems firstly are mathematically precise and that in itself is enough to put off a lot of people but they have the advantage of being sufficiently abstract. They specify the ‘what’ rather than the ‘how’ aspects of a language and there are no ambiguities. It is also an extremely concise method of specification. A lay person would probably call it too terse and completely incomprehensible but it is accurate and what you require is an accurate definition of the language.

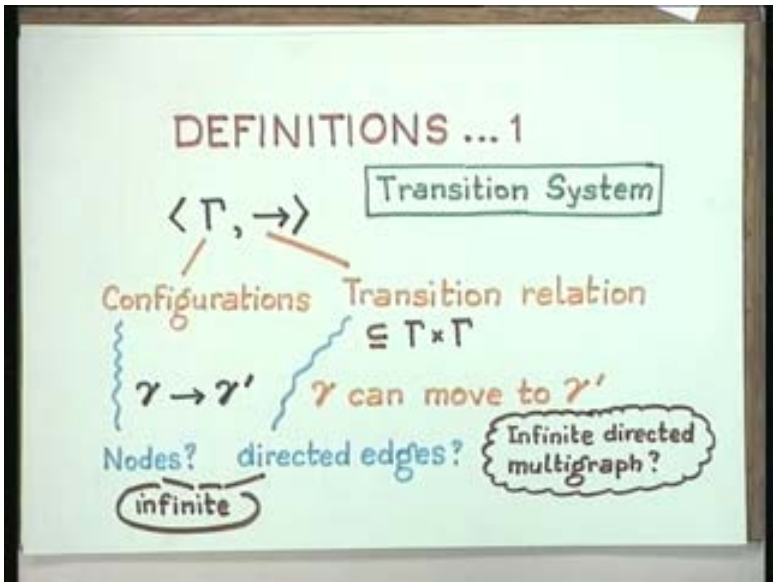
The other advantages of this particular method of transition systems is that it actually exploits definition by induction or definition by recursion and it uses induction on the structure of the syntax tree to the fullest extent to specify the meaning of any construct. It means that if you have a definition by induction then you can also use the principle of mathematical induction to reason about programs in this method. More important than anything else is the fact that actually it is not as intimidating as the lay person would think of it. It is actually quite general in many ways as a specification method because almost anything that you can think of could be expressed in terms of what are known as transition systems.

[Refer Slide Time: 15:06]



Its generality goes to such an extent that very common every day problems could also be specified as transition systems. For example; you could specify hardware as transition systems the move machines, the meeting machines etc are actually transition systems of various kinds. More importantly it allows you to abstract away from details. It allows an abstraction which is helpful in defining our equivalences. In fact you can be so abstract if you like with your transition system definition of a language that you could give very detailed analysis to the extent of capturing every step a runtime system will take or you can abstract away from all these details and give a larger picture which is denotational that allows you to define equivalences. You can take a purely functional view of whole programs or whole program units. So, transition systems are really quite general in that sense.

[Refer Slide Time: 18:57]



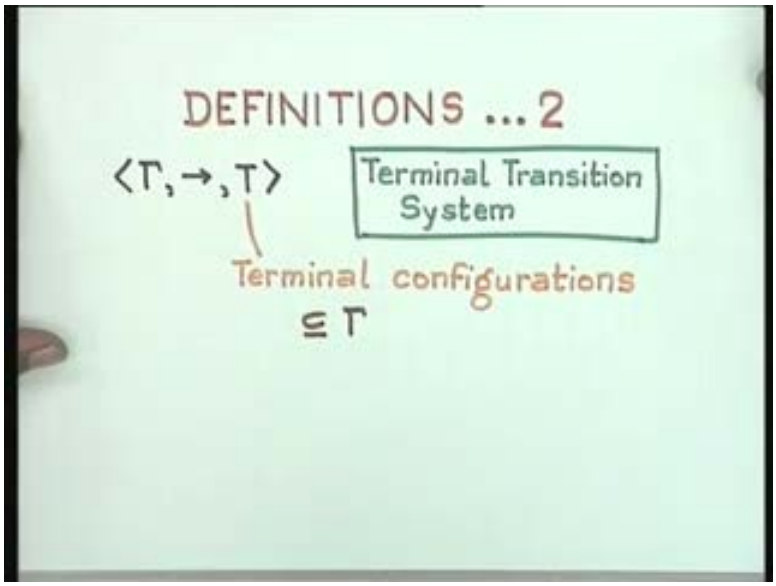
Let us go through some elementary definitions. What is a transition system? - A transition system is a structure consisting of a set gamma $< \Gamma, \rightarrow >$. This is the Greek letter capital gamma and an arrow relation where the gamma is called the set of configurations and the arrow relation is called the transition relation. This arrow is a binary relation on the configurations and we will usually use it in an infix form. So, given two configurations Γ and Γ', we will say that Γ can move to Γ'. The notion of a configuration is at this point not being specified but we will have to specify it when we actually get to the programming language.

The notion of a configuration in a particular programming language will also depend upon whether it is a declarative programming language or a functional language or an imperative language. The notions of configuration will differ depending upon what kind of language it is. But largely all imperative languages will have a similar notion of configuration. All functional languages would have a similar notion of configuration. All declarative languages like logic programming languages would have a similar notion of configuration. You can think about any transition system as essentially being some form of a multigraph except where the configurations are the nodes of the graph or the vertices of the graph. But the configurations need not be finite and neither is the transition relation finite. If you can think of allowing arbitrary kinds of directed infinitary multigraphs then you get a transition system.

In fact a directed graph can be thought of as another transition system and your representation of more than merely machines is really the form of directed graphs. With some special states, a special start state and a few special final states there are all these additions to the structure of a graph but essentially it is a graph. You can look upon programs also as essentially graphs and since only programs are capable of infinite behavior the graphs will have to be infinitary.

You can look upon programs as graphs in two different ways. You can look upon them as abstract trees which are also graphs. But it is of a purely syntactic matter. A program is a finite object in terms of its text so the syntax tree is also a finite directed graph. The runtime behavior of a program is such that it could go through an infinite number of states. The meaning of a program in terms of its behavior in execution could be a graph with an infinite number of nodes but directed all the same.

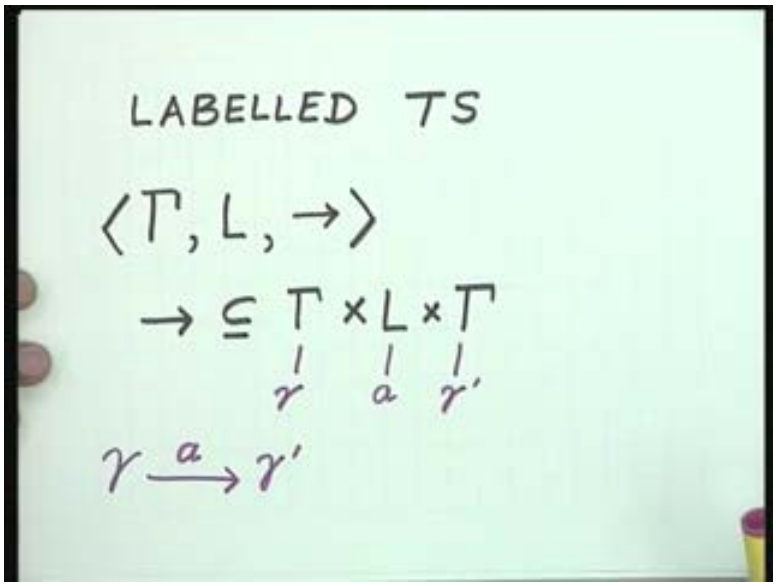
[Refer Slide Time: 22:00]



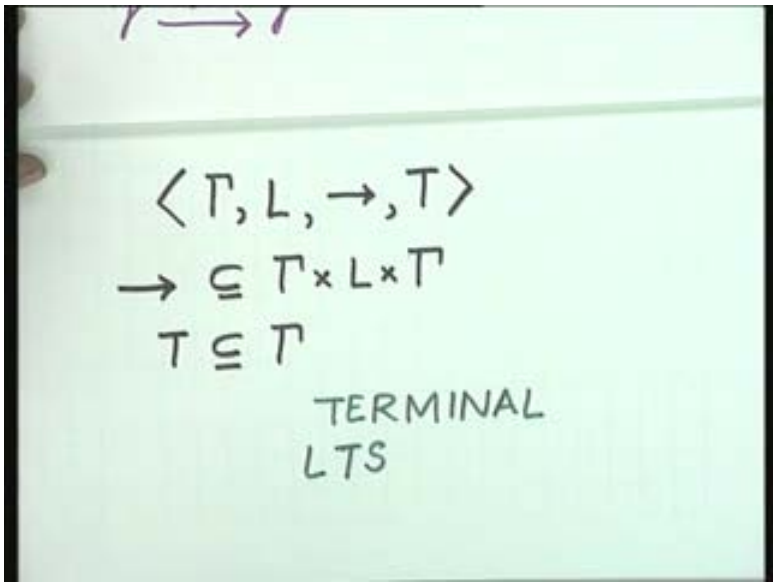
Our notion of the transition system is essentially the notion of a directed graph where we do not put any finiteness conditions on the nodes and the edges. Just as you can have label graphs where the edges carry labels with them, you could also define label transition systems. As in the case of some of the transition systems you have already done under various other names, we can also define a terminal transition system as a transition system in which there is a subset of configurations T, which are called halting configurations or terminal configurations.

You could go further and define a labeled transition system as something that also carries labels. There is a collection of configurations 'Γ', there is a set of labels which I will call L and there is a transition relation '$\rightarrow$'. In a three tuple, <Γ, L, $\rightarrow$> the set of labels need not necessarily be disjoint from the set of configurations but corresponding to the notion of a labeled directed graph, the transition relation is our ternary relation of the form; $< \Gamma, L \rightarrow > \rightarrow \subseteq \Gamma \times L \times \Gamma$ We would say that given a label let us say, 'a' and two transitions, Υ and Υ' we normally would write Υ can go on 'a' to Υ'.

[Refer Slide Time: 24:10]



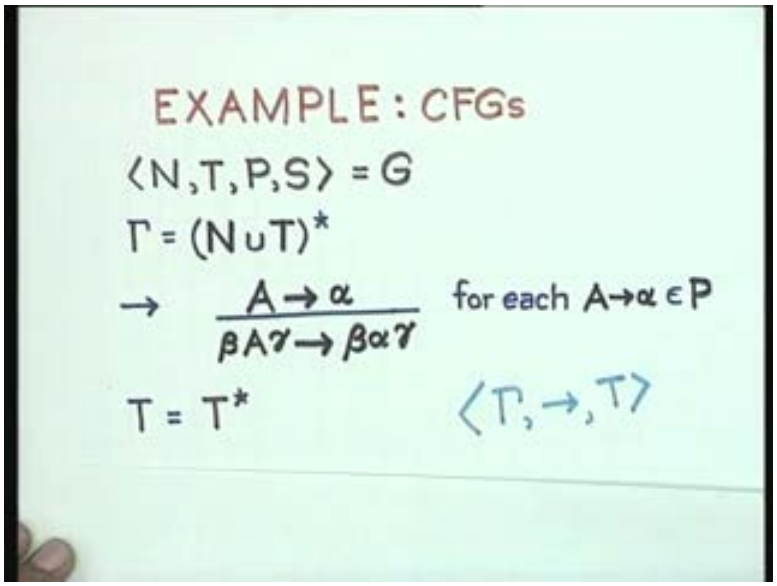
[Refer Slide Time: 25:06]



Correspondingly, you could also define terminal label transition systems. A terminal label transition system would just be a four tuple of this form $<\Gamma, L \rightarrow T>$ $\rightarrow \subseteq \Gamma \times L \times \Gamma$. This is a $T \subseteq \Gamma$ terminal labeled transition system and almost any kind of computational mechanism that you can think of can be represented in some kind of a transition system either just a transition system or a terminal transition system or a label transition system or a label terminal transition system and how you represent it in a transition system often might

depend on your view point. The fact that there exists a finite state machine which recognizes the sentences of a regular grammar that finite state machine is really nothing more than a label transition system. You can have finite state machines with outputs with every input which yields an output. Then your set of labels becomes slightly more complex consisting of input output pairs of symbols. This is a fairly general notion and actually we think of most of our computations in the form of transition systems. Let us take a simple example since you already know that regular grammars are equivalent to finite state machines. Let us give a transition system definition for context- free grammars. You can look upon it completely generally.

[Refer Slide Time: 30:22]



Let G be a grammar which consists of a collection of non terminal symbols, a set T of terminal symbols, a set of productions or rewrite rules or replacement rules and a start symbol S, $< N, T, P, S >= G$ and they are all finite. The set of configurations that we are talking about could just be the set of all strings from NUT, N union T, Γ = (NUT)* and the initial configuration is the symbol S. The transition relation in the transition system, Γ is just defined as for each production of the form 'A' that can be replaced by α in the production set, you have the rule that for any two strings β and Υ $\beta A \Upsilon$ can become $\beta \alpha \Upsilon$. The set of productions is finite so, the set of rules that you have is also finite where β and Υ are considered completely arbitrary.

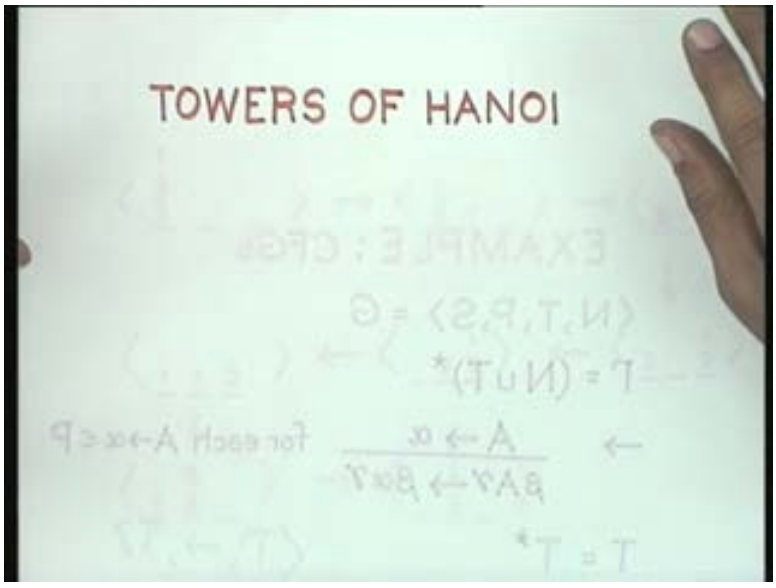
For any particular context-free grammar you can just think of it as a representation of the context-free grammar in the sense that the transition system defines exactly how your derivations should be done. It defines all possible ways of deriving terminal strings where by terminal I mean the set T of terminal symbols in the grammar. The set of terminal configurations in this transition system is just the set of all possible terminal strings. The terminal transition system is of the form $< \Gamma, \rightarrow T >$.

The right hand T of the transition system $< \Gamma, \rightarrow T >$ is just the set of all strings from the T in $< N, T, P, S >= G$ which is the set of terminals of the context-free grammar. Essentially, it is a terminal transition system so we require $\Gamma, \rightarrow T$ to be specified and the relation is completely specified by its rule which just says that for any B and any Γ, B$A\ \Gamma$ can go to B $\alpha\ \Gamma$ if 'A' replaced by α is a rule in the production set of this context-free grammar, P. If that is so then the arrow is a transition relation which gives you the set of all possible derivations in the context-free grammar. In the case of a context-free grammar, the symbol S has a special significance. You can add initial states also as part of a terminal transition system. It depends on what kinds of details and distinctions you have to add but essentially the basic framework is that of transition systems.

You could actually go further. If you take the ruler-compass constructions in geometry you can look upon each ruler-compass construction method itself as an algorithm. Let us take a simple problem like constructing a regular hexagon. The method starts with taking a circle of a certain radius, choosing an arbitrary point in the circle and with that same radius marking out the 6 vertices of the hexagon and completing the hexagon. Each of these stages is like a transition. The start state in this case is a blank piece of paper and then you move to a circle, then you move through six steps to marking out the potential vertices of the hexagon and then you move through six steps in completing the hexagon.

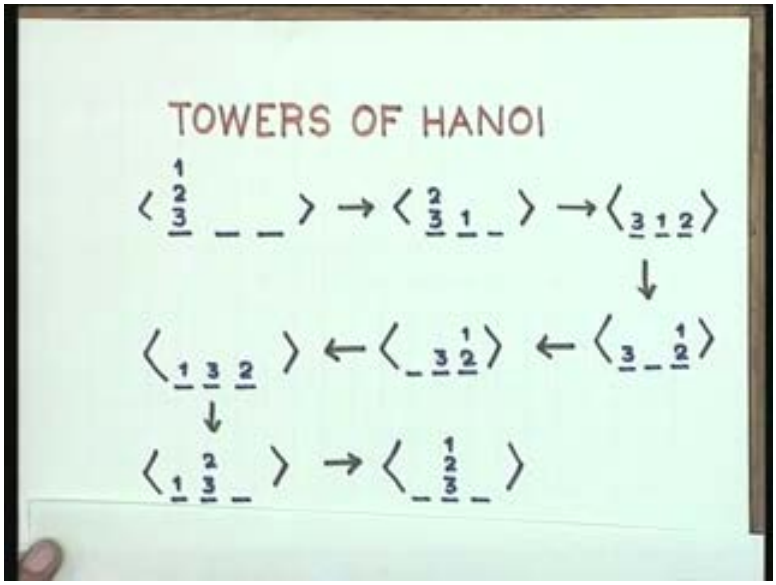
They are all transitions which are the steps of an algorithm being executed. But if you look at all of these transitions as snap shots in the process, it just specifies 'what' and it does not specify 'how'. The level of detail could be so high that the question of how might be implicit but essentially it just specifies 'what'. If I just take these let us say, fifteen snap shots of the construction of a regular hexagon, it just specifies 'what' is to be specified. It just specifies transitions.

[Refer Slide Time: 34:20]



Let us take a simple programming example. Here is the tower of Hanoi problem. You have got three towers and you have got some n pegs all of different sizes and the rules normally stated very solemnly are that all the pegs are of different sizes and a larger peg cannot sit on top of a smaller peg. Note the word “cannot”. Another rule that is stated is that if you have got a pile of pegs then you can move only the top peg. You cannot move any other peg from inside the pile and the problem is given that these are the rules, by various moves to get this set starting from an initial configuration in which the n pegs are stored one above the other such that smaller pegs always sit on the larger pegs, you have to move that entire collection to another tower. Let us quickly go through one execution of this.

[Refer Slide Time: 37:24]

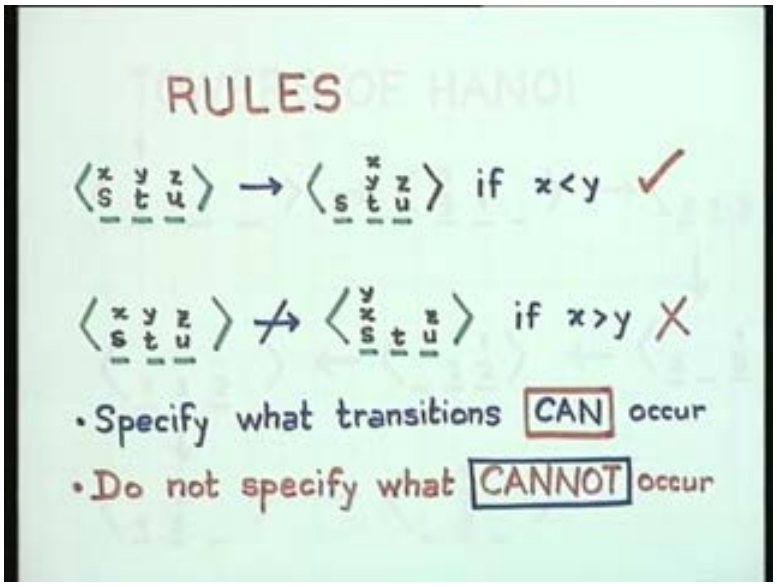


Here you have some initial configuration. These are the three towers and the initial configuration is that you have these three pegs which are numbered according to their size sitting on the first tower and the final step is to move them all to the second tower. You can go through a sequence of transitions. Take the top peg in the first tower and put it on the second tower. You can move the top peg in the first and move it to the third and take the peg from the middle tower and put it on the third. Take the peg from the first tower and move it to the second, take the top peg from the third tower and move it to the first, take the peg from the third tower and move it to the second and take the peg from the first tower and move it to the second.

This is a step by step specification of an actual execution of towers of Hanoi program of one possible algorithm. The question is how do you formulate rules? For example; this execution does not tell you that you could not have moved this 3 on top of the 2 or on top of the 1. It is important for us not to specify complete execution behaviors because a programming language has an infinite number of possible programs and you cannot specify their execution behaviors completely.

You have to specify the rules and you have to claim that any particular execution follows those rules. The productions of the context-free grammar essentially are rules or at least in our definition of the context-free grammar, as a transition system, they act as a hypothesis of rules which give you how derivations are performed. Note that they actually tell you how derivations are to be performed but they do not tell you how derivations should not be performed. In the way we stated the problem of towers of Hanoi and the way it is stated in most books, it is specified what you cannot do which is an integral part of the problem but in most of mathematics nobody ever tells you what you cannot infer. They only tell you what you can infer. It is understood that you do not have a theorem of impossibility unless you can show that it is impossible to prove that theorem with what you can do.

[Refer Slide Time: 42:23]



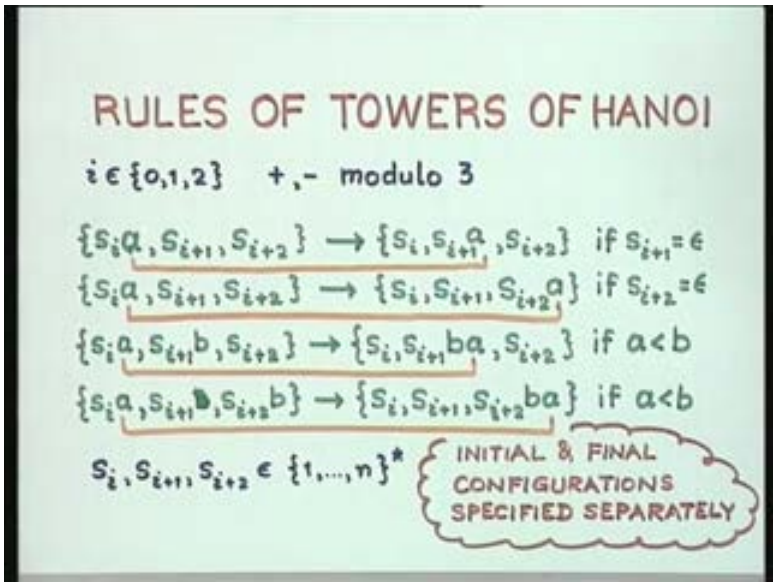
This is absolutely important. In our natural language specification we specify what you cannot do and what I claim is that in most of mathematics what you cannot do is never specified explicitly. We have to design the specification rules for the towers of Hanoi problem without stating what you 'cannot' do. You should only specify what you can do. That means that you could specify a rule like the following. If I had a sequence of pegs let us say s. There is some sequence s, there is some x, there is some t and there is a y on top, there is a u and then there is a z on top. In it 'x, y and z' denote the top most pegs and's, t and u' denote arbitrary strings in the pegs. So, you can specify a rule which says that if x<y then the peg can be moved on top of the y. But you cannot specify rules like this. Our natural language definition of the problem said that you cannot for example, move a larger peg on top of a smaller peg and this is essentially a symbolization of that. But you should define the rules in such a way that you do not carry out the entire process stated. After all what you cannot do with a program is really infinite. What is finite is only what you can do with a program.

If you are going to keep specifying what all you cannot do, you have an infinite number of rules and it is not clear that you will ever proceed with execution. Our rules will always specify what can be done and never what cannot be done. This is perfectly in keeping with logical inference. You do not have a rule which says that from ‘a’ and ‘a and b’ you cannot infer ‘not a’. You do not have such rules. The fact that you cannot infer ‘not a’ follows from the notion of consistency or inconsistency of a logical system, or of an axiomatic system. A logical system just specifies what you can infer and never what you cannot infer. What you cannot infer occurs as an impossibility of proof. We will follow this principle also in our specification of the language.

Note that pragmatically speaking also it is a nice way of specifying things. Anything that goes against what you can do is an error. There are of course certain cases in natural language where this kind of reasoning does not work. In natural language you have to specify what you cannot do but when you are talking about programming languages and formal objects such as programs and algorithms they are all formal objects and very highly structured mathematical objects. Although algorithms are not very formal they have their representation as programs. So, it is possible to get by with what you can do and assume that whatever you cannot do follows as some larger consequences of what all you can do.

If I were to give rules for the towers of Hanoi as a transition system, I will not exactly specify all the possible configurations. I will just specify the arrow relation, the transition relation and it is convenient to have a concise, precise and small set of rules if possible. I will take an index ‘i’ from 0, 1 and 2. The index ‘i’ could be arbitrary belonging to this set and whenever I deal with addition and subtraction it is always modulo 3. The three towers are actual executions specified in order for the towers. It does not matter in what order I keep these three towers. If I name them 0, 1 and 2, then I can use modulo 3 arithmetic to just call them i, i + 1 and i + 2. I could take values from 0, 1 or 2 with modulo 3. Then I can look upon the contents of the three towers as a set of three elements and the set of all such possible sets is my set of configurations.

[Refer Slide Time: 49:05]

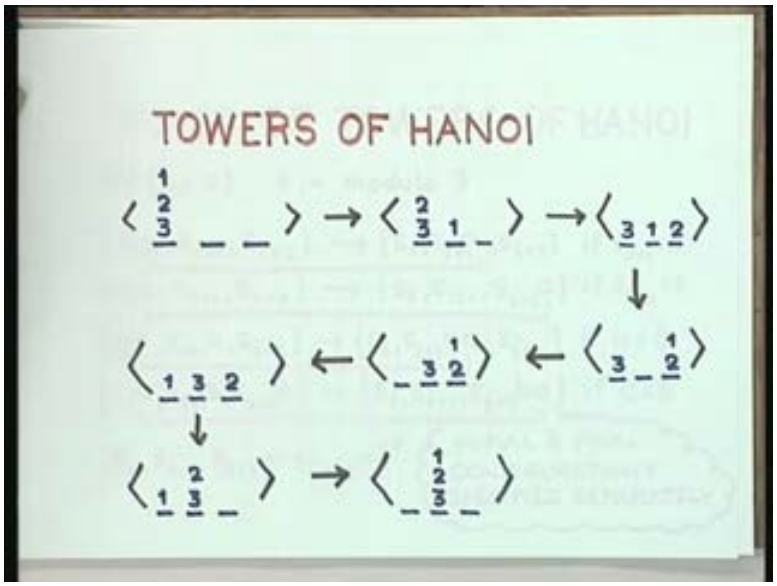


There are specifications about what an initial configuration is and what a terminal configuration is but that is part of the problem. It goes beyond the notion of the transition. Regardless of what ‘i’ is and regardless of what order the towers are placed I can specify the rule. The first rule says that supposing the tower i, (whatever may be i) is non empty that means it has at least one, the top peg is called ‘a’ and i + 1 is empty. If the tower i + 1 is empty then the top peg ‘a’ from i can be moved to i + 1. Similarly, if i + 2 is empty the top peg ‘a’ could be moved to i + 2. Note here that both are allowed which means starting from some initial configuration it is not clear which one you are going to do. You could do either of them.

For example; if you had one tower which contains all the pegs and the other two towers are empty, either of the rules can be applied. You can move the top peg either to one tower or to the other. It is important to say that the arrow relation in a transition system does not specify what does happen or what should happen but it specifies what can happen. It does not preclude other possibilities.

Let us go through the next one. This is like a base case of an induction because you cannot compare the top element of an empty tower with a top element of a non empty tower. If a < b and b is a top element of the second tower then a can be moved on top of b. Similarly, if a < b and b is a top element of the first tower then a can be moved on top of b. I have not specified anywhere what cannot be done. These rules specify exactly what can be done. Given an initial configuration you can verify that we have got this execution.

[Refer Slide Time: 49:19]



Each transition here is an application of one of the four rules. So, that is what is important about transition systems and it is in fact part of any kind of mathematical subject. You never specify what might be loosely called negative facts. You can even take some purely functional program like a factorial program. The factorial program, by definition is a mathematical equality but the unfolding of a recursion can be regarded as a one-way rewrite rule or a transition. You can take the definition of the factorial program and go through a sequence of transitions. Any given factorial of n will go through a sequence of transitions which give you different configurations. The notion of a configuration might be an expression itself which is the unfolding of the recursion.