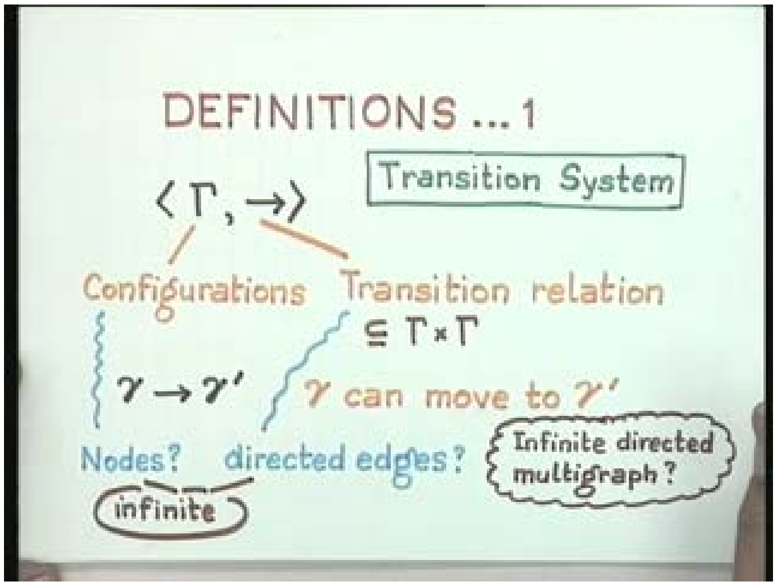
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 9
Lecture Title: PL0: Expressions

Welcome to lecture 9. We will define the semantics of expressions of PL0. We will take a subset and look at it in detail before we proceed. Let me also briefly recapitulate the notion of transition systems which we are going to use.

[Refer Slide Time: 00:51]



DEFINITIONS ... 1

Transition System

$\langle \Gamma, \rightarrow \rangle$

Configurations — Transition relation $\subseteq \Gamma \times \Gamma$

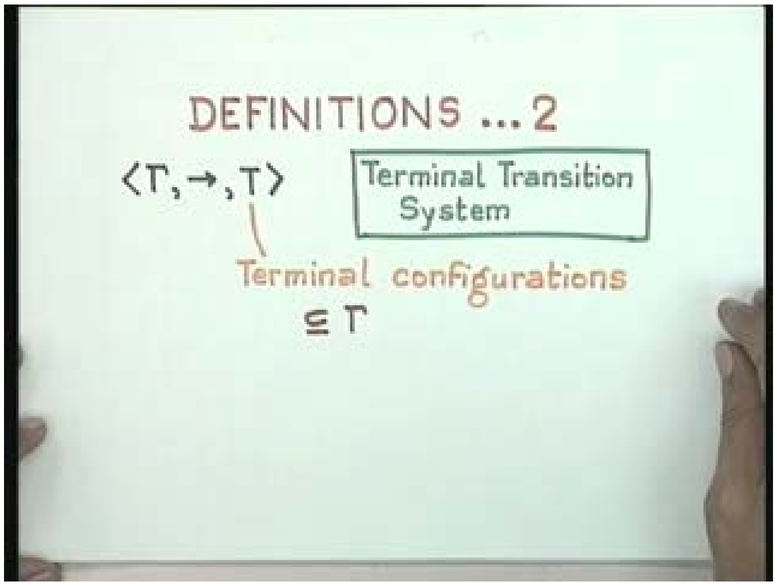
$\gamma \rightarrow \gamma'$ — γ can move to γ'

Nodes? directed edges? Infinite directed multigraph?

infinite

A transition system is just a collection of configurations along with a binary relation called the transition relation and the transition relation really tells you what is possible and not necessarily what actually happens. We could regard a transition system as some kind of an infinite directed multigraph in general. We could define initial states, final states, final configurations, initial configurations and enhance the distinguishing power of a transition system. A terminal transition system would typically have a collection of terminal configurations which are a subset of the configurations.

[Refer Slide Time: 01:28]



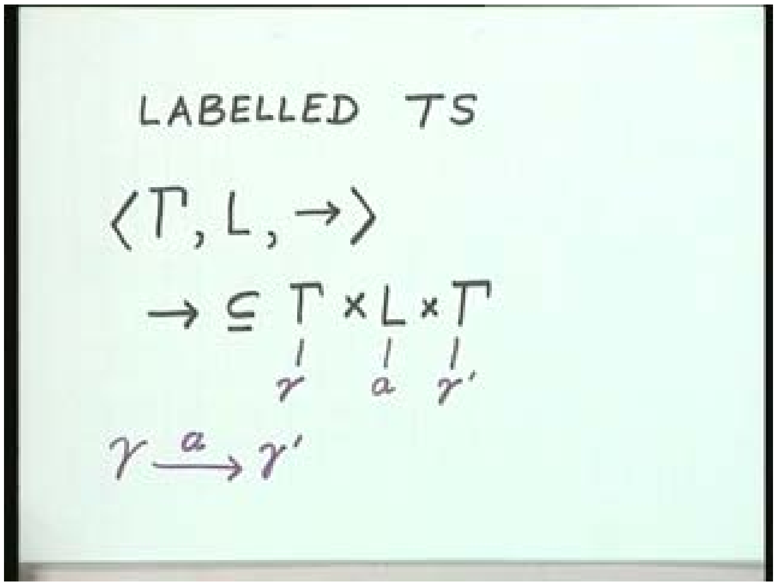
DEFINITIONS ... 2

$\langle \Gamma, \rightarrow, T \rangle$ $\boxed{\text{Terminal Transition System}}$

Terminal configurations $\subseteq \Gamma$

You could also decorate the transition system further depending upon the use you are putting it to. You could for example define label transition systems which in addition to being transition systems have a collection of labels associated with them.

[Refer Slide Time: 02:00]



LABELLED TS

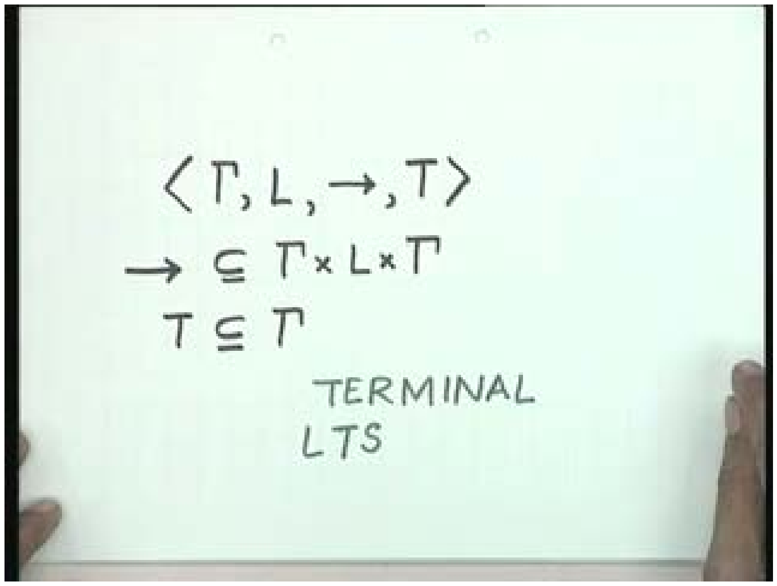
$\langle \Gamma, L, \rightarrow \rangle$

$\rightarrow \subseteq \underset{\gamma}{\Gamma} \times \underset{a}{L} \times \underset{\gamma'}{\Gamma}$

$\gamma \xrightarrow{a} \gamma'$

This would correspond to labeled infinite directed-multigraphs. There are arcs of labels on them. The transition relation therefore is a ternary relation. If L is a set of labels then you would say that Γ can move to Γ' on 'a'. The notion of configuration and of labels is loosely specified. In fact they are really undefined terms that make the whole structure generally applied. You could interpret the notion of the configuration to suit your convenience. You could interpret what constitutes a label to suit your convenience and that is what makes transition systems a general mathematical tool for specification. You could enhance them further.

[Refer Slide Time: 03:25]



$\langle \Gamma, L, \rightarrow, T \rangle$

$\rightarrow \subseteq \Gamma \times L \times \Gamma$

$T \subseteq \Gamma$

TERMINAL

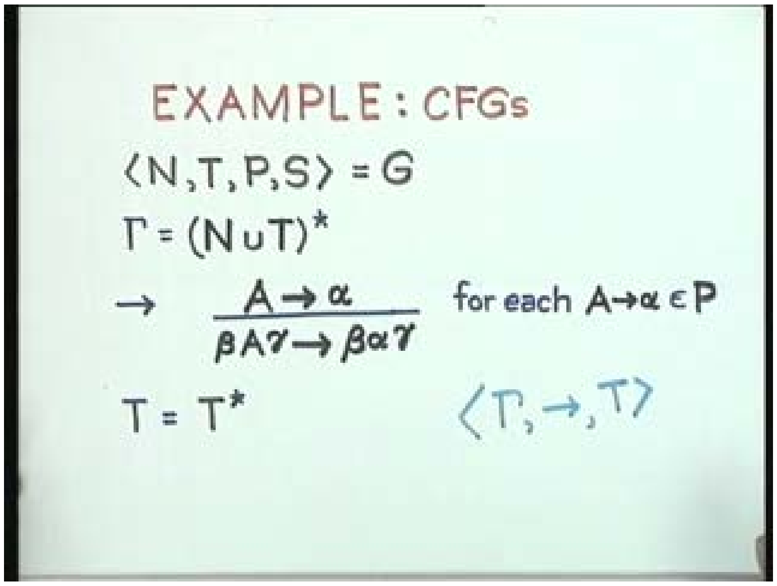
LTS

You could define terminal label transition systems. You could define label transition systems with initial configurations, terminal label transition systems with initial configurations etc. But the basic idea is that of movement from one configuration to another and depending on your distinguishing capability you could enhance or decorate them with more constraints.We already saw one example of context-free grammars represented as a transition system with a rule which really says that it is a context-free rule of replacement.

For any β and Υ $\dfrac{A \rightarrow \alpha}{\beta A \Upsilon \rightarrow \beta \alpha \Upsilon}$

We will use this notation which actually comes from logic but the way to look at it is as a rule which gives a hypothesis and what you can conclude. If 'A -> a' is a production in the production set then B A Γ -> B a Γ is a step in a transition or in a derivation.

[Refer Slide Time: 04:00]



EXAMPLE : CFGs

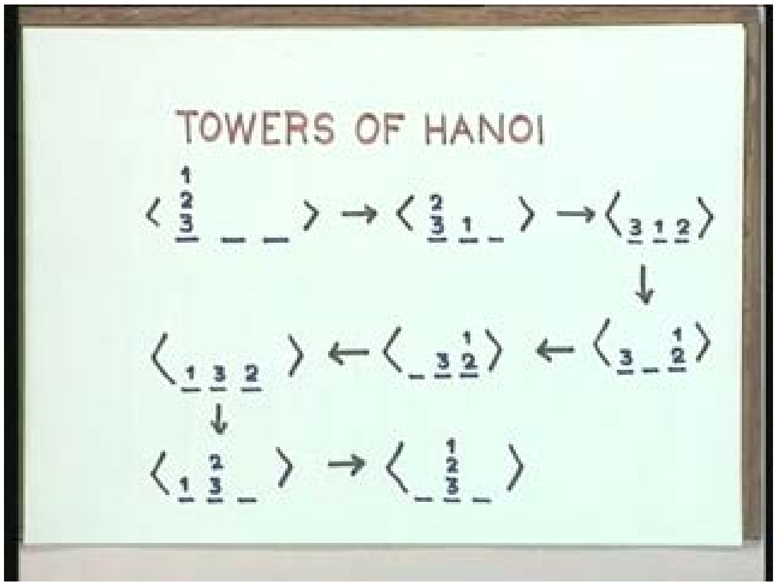
$\langle N, T, P, S \rangle = G$

$\Gamma = (N \cup T)^*$

$\rightarrow \quad \dfrac{A \rightarrow \alpha}{\beta A \gamma \rightarrow \beta \alpha \gamma}$ for each $A \rightarrow \alpha \in P$

$T = T^*$ $\quad \langle \Gamma, \rightarrow, T \rangle$

[Refer Slide Time: 05:10]



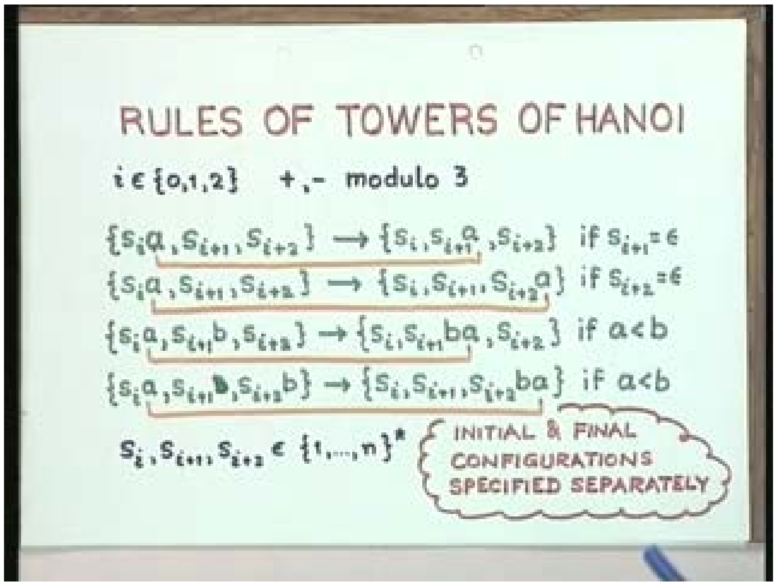
TOWERS OF HANOI

We also looked at the towers of Hanoi problem. In this case we saw how ‘a particular execution’ of the towers of Hanoi problem would look like. It really specifies the runtime behavior of some particular algorithm which is actually unspecified. It is an execution behavior and you use the transition system to define the execution behavior of some algorithm or a whole class of algorithms which support this possible sequence of transitions. Not all correct algorithms for the problem might actually support it but there is a whole class of algorithms which support it.

We would like to abstract away the essentials of what allows these transitions. The problem of ‘what’ is to be specified as opposed to the problem of ‘how’ to specify is what we want to abstract away by the rules. The problem itself can be stated as a collection of rules. The execution behavior can be stated as a collection of rules. Given an algorithm you can follow its execution and represent that execution as a transition system. Given a problem you can look upon the problem also as a transition system provided you give suitable rules.

This is not one possible solution of the towers of Hanoi problem because the problem itself is to design an algorithm which will allow you to move n pegs from one tower to another. The problem is just that given this initial configuration how do you get this final configuration? This is just one possible execution of a few algorithms. The actual rules themselves of the problem, which again do not constitute an algorithm, constitute a statement of the problem as given. They essentially say that you can move the peg on the top of a tower to an empty tower to either of the two towers.

[Refer Slide Time: 07:45]



RULES OF TOWERS OF HANOI

$i \in \{0,1,2\}$ $+,-$ modulo 3

$$\{s_i a, s_{i+1}, s_{i+2}\} \rightarrow \{s_i, s_{i+1} a, s_{i+2}\} \text{ if } s_{i+1} = \epsilon$$

$$\{s_i a, s_{i+1}, s_{i+2}\} \rightarrow \{s_i, s_{i+1}, s_{i+2} a\} \text{ if } s_{i+2} = \epsilon$$

$$\{s_i a, s_{i+1} b, s_{i+2}\} \rightarrow \{s_i, s_{i+1} ba, s_{i+2}\} \text{ if } a < b$$

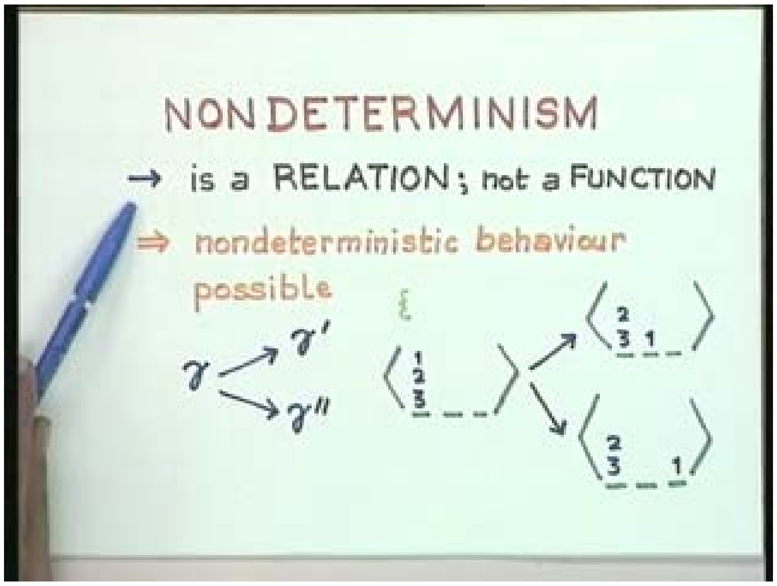
$$\{s_i a, s_{i+1} b, s_{i+2} b\} \rightarrow \{s_i, s_{i+1}, s_{i+2} ba\} \text{ if } a < b$$

$s_i, s_{i+1}, s_{i+2} \in \{1, \ldots, n\}^*$

INITIAL & FINAL CONFIGURATIONS SPECIFIED SEPARATELY

The two towers might be empty so you could move the peg to either of them. Similarly, if a is less than b and a and b are on different towers then you could move ‘a’ on top of b. The two rules include the fact that if you have s i a, si+1 b, si + 2 c where a<b and a<c then you could apply any of the two rules and put ‘a’ either on top of b or on top of c. Our transition systems are inherently non deterministic and that is because the transition relation is really a relation and not a function.

[Refer Slide Time: 09:00]



NON DETERMINISM

→ is a RELATION ; not a FUNCTION

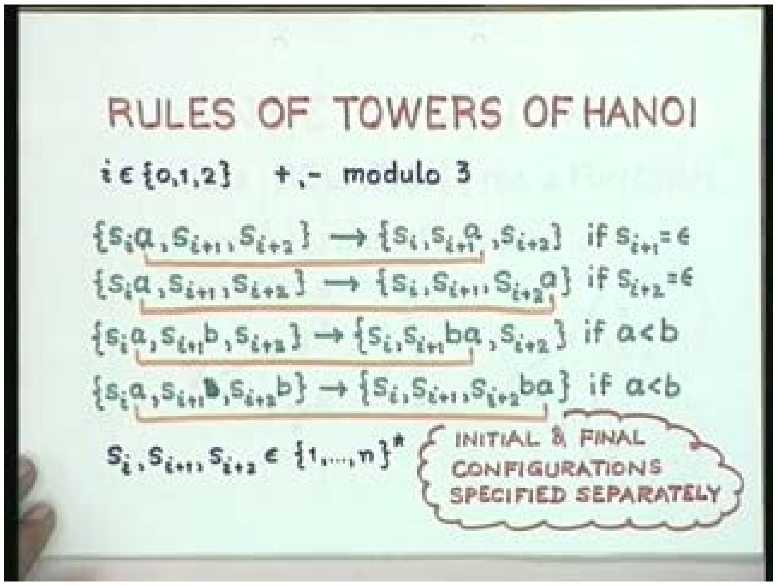
⇒ nondeterministic behaviour possible

$$\begin{tikzcd} & \gamma' \\ \gamma \arrow[ur] \arrow[dr] & \\ & \gamma'' \end{tikzcd}$$

If it were a function then the possibility of a configuration moving to two distinct configurations would not be allowed. The execution that we have chosen is one possibility out of a variety of different possible executions for a whole class of algorithms. Moreover non determinism at least in a large part of computer science is merely a form of under specification in the sense that you are not extremely concerned sometimes about whether you pursue one path or another. In the statement of the problem it does not specify to you that from the tower ‘i’ a peg can be moved only to tower i+1 and it cannot be moved to tower i+2. Nowhere in the problem is that stated and is never stated. As a result the problem statement itself is such that it is under specified so that it allows you a variety of possible solutions and non determinism is an implicit fact of life in any kind of succinct problem definition.

In many cases it is also a fact of life of the solution because you are interested sometimes not in what order you perform certain things. If I asked you to find all the prime factors of a number then some $2^4 \times 3^6$ it really does not matter in what order you find the four 2’s and the six 3’s. You could define a solution in which you find a first 2 and then you find a first 3, go back find two more 2’s and find three more 3’s. The fact that you write an algorithm in a deterministic programming language such that it might first find all the tool factors; it might just factor out the 2’s and then start with all the 3’s but that is just one possible solution. There is nothing to prevent you from having found a 2 and then finding 3’s and then go back to finding some more 2’s. Even your solution’s pairs could be non deterministic depending on the level of detail you want to put in.

[Refer Slide Time: 12:35]



RULES OF TOWERS OF HANOI

$i \in \{0,1,2\}$ $+,-$ modulo 3

$\{s_i a, s_{i+1}, s_{i+2}\} \rightarrow \{s_i, s_{i+1} a, s_{i+2}\}$ if $s_{i+1} = \epsilon$

$\{s_i a, s_{i+1}, s_{i+2}\} \rightarrow \{s_i, s_{i+1}, s_{i+2} a\}$ if $s_{i+2} = \epsilon$

$\{s_i a, s_{i+1} b, s_{i+2}\} \rightarrow \{s_i, s_{i+1} ba, s_{i+2}\}$ if $a < b$

$\{s_i a, s_{i+1} b, s_{i+2} b\} \rightarrow \{s_i, s_{i+1}, s_{i+2} ba\}$ if $a < b$

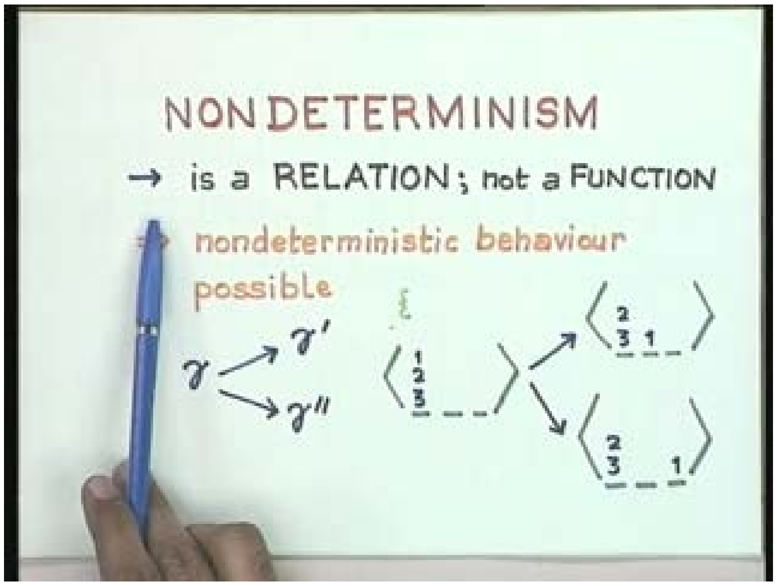
$s_i, s_{i+1}, s_{i+2} \in \{1, \ldots, n\}^*$

INITIAL & FINAL CONFIGURATIONS SPECIFIED SEPARATELY

Non determinism is an intrinsic fact of life both of problems and of classes of solutions for the problem. The problem statement in the case of towers of Hanoi is clearly non-deterministic because it allows the application of one of two possible rules at each go. In fact if it so happens that one tower is empty and another tower has an element ‘b’ at the top which is larger than ‘a’. Then at any time two out of the four rules could be applied. So, in that sense it is distinctly non deterministic.

If you were to generalize the problem to include many more towers then you might at some point find that all four rules could be applied but you always choose one possible rule but that is not a unique solution. What is important is that since transition systems are a general formalism they allow non determinism. However, all the programming languages that we use are intrinsically deterministic which means in your semantics you have to show that there are only deterministic behaviors. For a given program there is only one possible behavior and showing that means proving that your transition relation is a function. Very often you will have to show this by induction on the structure because we are going to use the syntactic structure to a great extent.

[Refer Slide Time: 14:20]



NONDETERMINISM

→ is a RELATION; not a FUNCTION

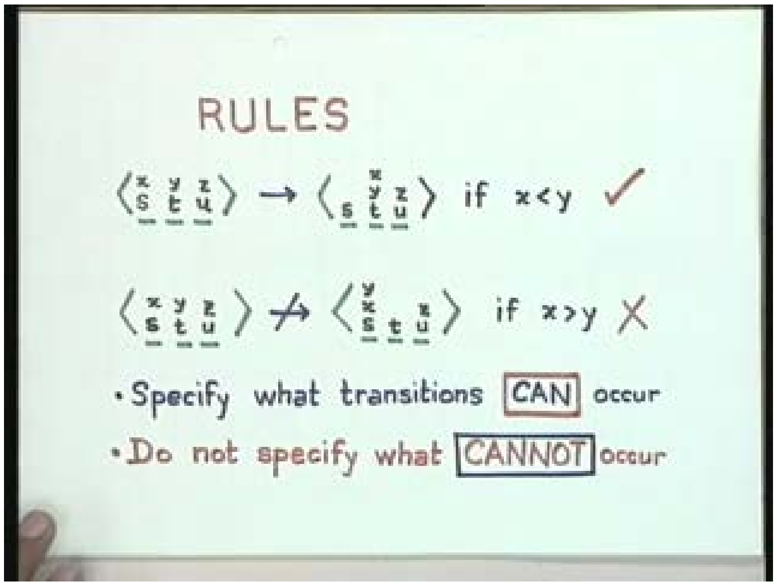
⇒ nondeterministic behaviour possible

$$\begin{tikzcd} & \gamma' \\ \gamma \arrow[ur] \arrow[dr] & \\ & \gamma'' \end{tikzcd}$$

$$\begin{tikzcd} & \left\langle \underline{\substack{2\\3}}\ \underline{1}\ \underline{\ } \right\rangle \\ \left\langle \underline{\substack{1\\2\\3}}\ \underline{\ }\ \underline{\ } \right\rangle \arrow[ur] \arrow[dr] & \\ & \left\langle \underline{\substack{2\\3}}\ \underline{\ }\ \underline{1} \right\rangle \end{tikzcd}$$

Next, as I said, whenever you specify rules, you specify what might be loosely termed positive rules and no negative rules. You never specify what cannot be done or what transitions are not possible. Whatever transitions are not possible is inferred by default if you can prove that with the existing rules it is not possible to get a transition of this form. It is always inferred and you do not specify it. You always specify what are usually called positive rules. You do not specify what cannot happen.

[Refer Slide Time: 14:40]



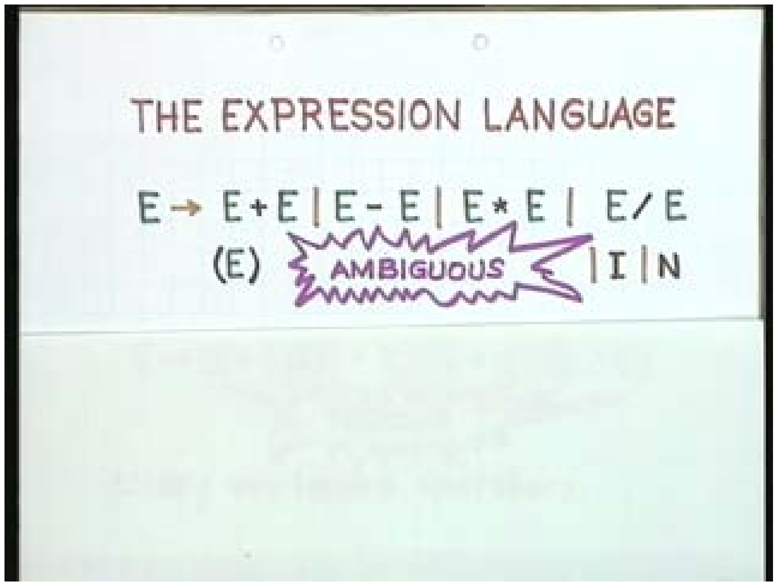
RULES

$\left\langle \underline{\substack{x\\s}}\ \underline{\substack{y\\t}}\ \underline{\substack{z\\u}} \right\rangle \rightarrow \left\langle \underline{s}\ \underline{\substack{x\\y\\t}}\ \underline{\substack{z\\u}} \right\rangle$ if $x<y$ ✓

$\left\langle \underline{\substack{x\\s}}\ \underline{\substack{y\\t}}\ \underline{\substack{z\\u}} \right\rangle \not\rightarrow \left\langle \underline{\substack{y\\x\\s}}\ \underline{t}\ \underline{\substack{z\\u}} \right\rangle$ if $x>y$ ✗

- Specify what transitions $\boxed{\text{CAN}}$ occur
- Do not specify what $\boxed{\text{CANNOT}}$ occur

[Refer Slide Time: 15:40]



THE EXPRESSION LANGUAGE

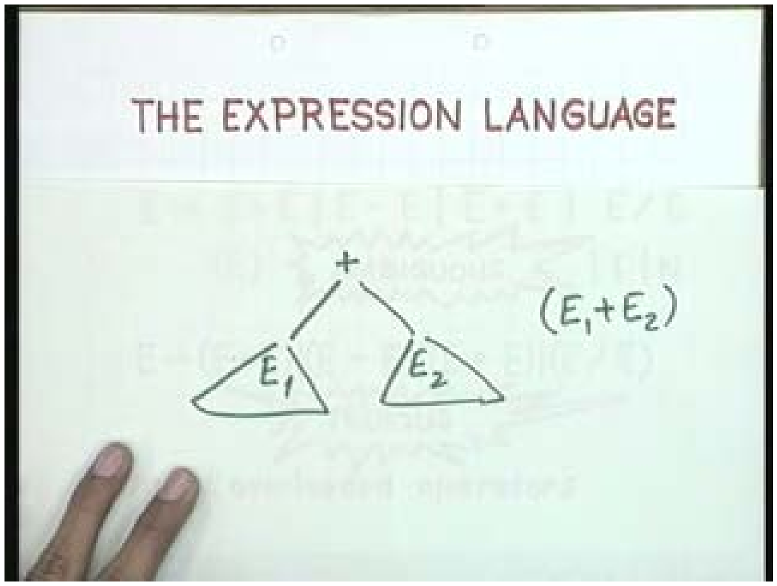
$E \rightarrow E+E \mid E-E \mid E*E \mid E/E$

(E) AMBIGUOUS $\mid I \mid N$

Let us look at the expression language of PL0. I had previously given this syntax which I called ambiguous. We know how to get rid of ambiguity for the purposes of parsing and compiling but we are interested in specifying a semantics that is not cluttered up with all those non terminals which do not have great semantic significance. We will deal with this ambiguous grammar but whenever we write an expression it is understood that we are writing not an expression but a tree. We are really interested in the syntax trees in an abstract form and we are not really interested in what might be called the lexical properties of whatever grammar we write. This way the grammar is more succinct, more precise and this way the grammar is actually part of what is know as term algebra.

I am going to do induction on the structure so if you want you can assume that I have a fully parenthesized notation to get rid of ambiguity. If your grammar is inherently ambiguous then you have more than one syntax tree for an expression and that can change the meaning. You can transform that syntax tree into a fully parenthesized notation using brackets. The rules of semantic specification are going to be based on syntax trees in the sense that they are going to be inductive rules based on the root operator of the expression or of the syntactic category. You can look upon the entire expression either as a fully parenthesized expression of the form let us say (E + E) or you can look upon a tree whose root is a plus operator and it has two sub trees which let us say are called E 1 and E 2.

[Refer Slide Time: 20:49]

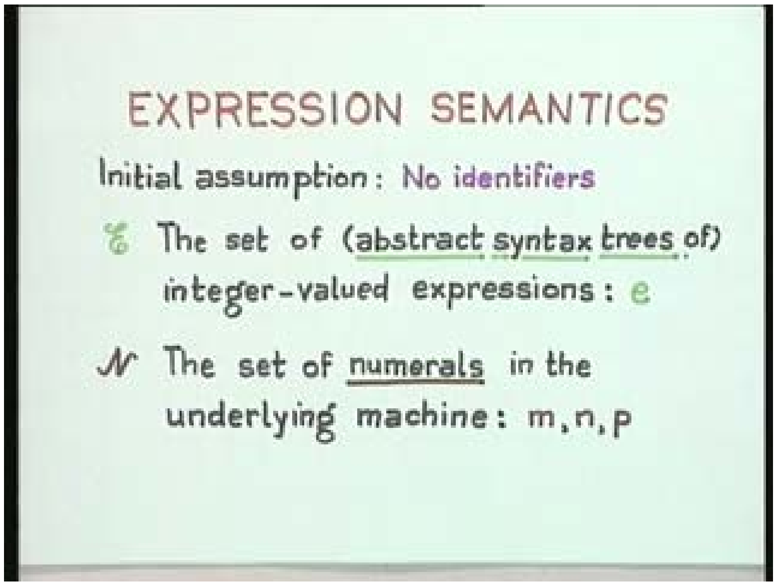


You can think of each of the expressions as being specified in this form where E 1 and E 2 themselves are trees. That is how you do induction on trees.

How is a tree defined? A single node is a tree given two trees t 1 and t 2 and a node with some label on it, the tree formed by t 1 as left sub tree and t 2 as right sub tree with that node as a root is a tree. Given E 1 and E 2 as trees with a plus operator as a root you get a tree. So, trees are defined inductively and we will look at the inductive structure of the trees and not at the actual lexical syntax. If you want to clarify the lexical syntax for yourself, you can look upon it as representing the expression $(E_1 + E_2)$ fully parenthesized. We are just interested in expressions as a category. Let us assume that we have got a collection script e. It is just the set of all abstract syntax trees of integer valued expressions. For the present I am assuming that there are no identifiers because we can deal with identifiers only after we have come up with a notion of an environment.

We will do that later but for the present just assume an expression language which does not contain any identifiers or this production since we are not interested in bracketing and it does not contain this. Let us go back to PL0 identifiers. The identifiers in an expression could either be variable names or they could be constant names but now when I get rid of these two productions I just have a pure expression language expressed in terms of the numerals. The roots or the leaves of all the trees are just numerals. That is my initial assumption because identifiers really require the notion of an environment. Let us give a simple transition system for a language of just expressions on numerals.

[Refer Slide Time: 21:30]



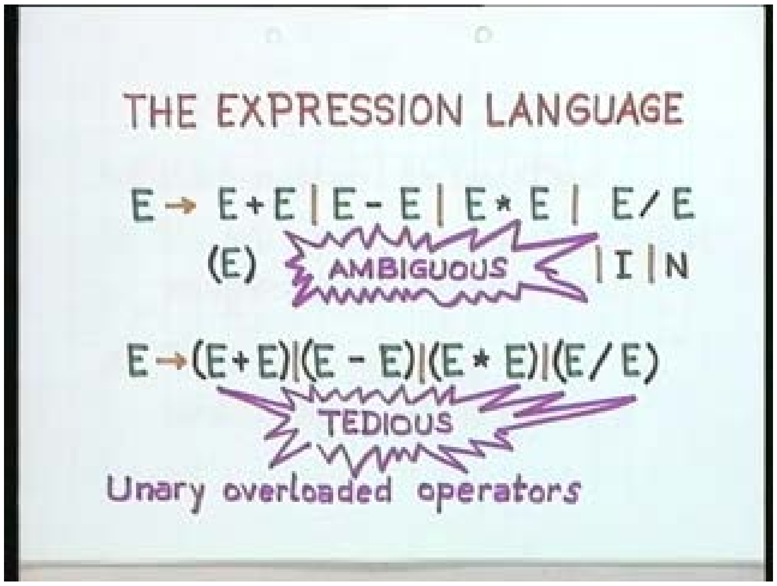
EXPRESSION SEMANTICS

Initial assumption: No identifiers

$\mathcal{E}$ The set of (abstract syntax trees of) integer-valued expressions: e

$\mathcal{N}$ The set of numerals in the underlying machine: m, n, p

[Refer Slide Time: 22:14]



THE EXPRESSION LANGUAGE

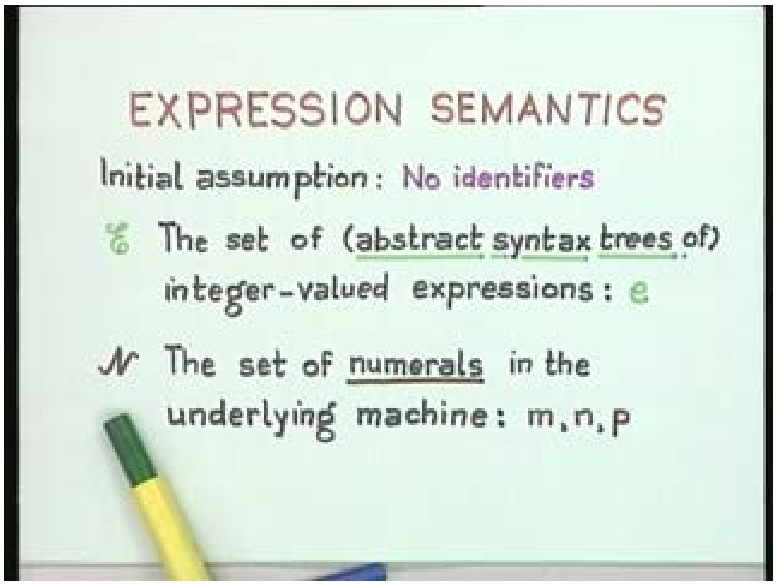
$E \rightarrow E+E \mid E-E \mid E*E \mid E/E$
(E) AMBIGUOUS $\mid I \mid N$

$E \rightarrow (E+E) \mid (E-E) \mid (E*E) \mid (E/E)$

TEDIOUS

Unary overloaded operators

[Refer Slide Time: 23:30]



EXPRESSION SEMANTICS

Initial assumption : No identifiers

$\mathscr{E}$ The set of (abstract syntax trees of) integer-valued expressions : e

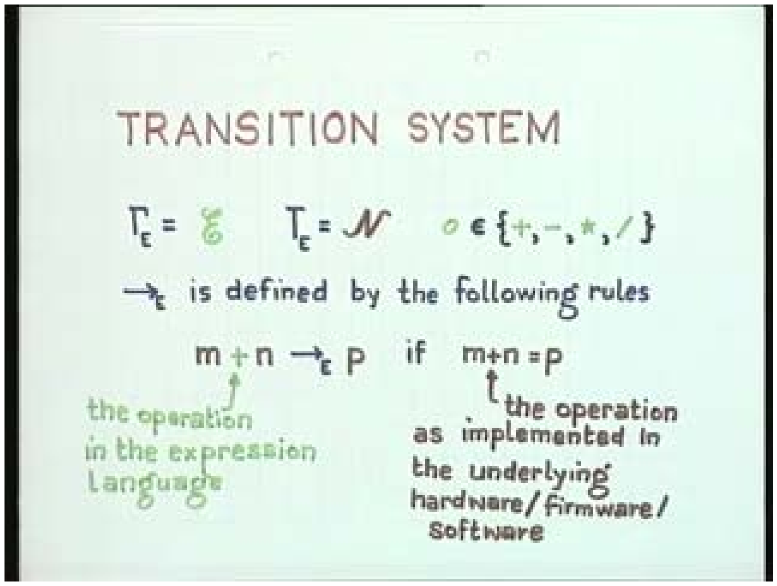
$\mathcal{N}$ The set of numerals in the underlying machine : m, n, p

The N denotes the set of all numerals in the underlying machine. The underlying machine could be a virtual machine which means that as in the case of PL0 compiler the PL0 compiler is written in Pascal so the virtual machine that you have got is a Pascal machine. It is a bare machine with layers of software and the outermost layer, which forms the interface for the PL0 compiler is a Pascal machine. So, it gives the view that what you have got underneath is just a Pascal machine with all the capabilities of Pascal and nothing else.

This underlying machine might be either real or virtual. I am not interested in its architecture. I am not interested in the details of the machine except in so far as it gives me certain computational primitives that I can use. In the case of an expression language the basic computational primitives that you might require are addition of integers, subtraction of integers, multiplication of integers etc. Assume that those basic operations are somehow implemented either in hardware or in software in the underlying machine which itself might be either virtual or real.

The transition system for this language with no identifiers that just works on numerals consists of a collection of configurations Γ_e which is just the script e. The terminal configurations are all the numerals, N and we have $oe\{+,-,*,/\}$ where the little 'o' to denote a binary operation. Our underlying computational model assumes that the operations are already available. They are anyway already usually available in hardware. So, I do not make any assumptions about the architecture except that I am assuming that the corresponding operations are already available in the underlying machine and that greatly simplifies my transition system.

[Refer Slide Time: 29:52]



TRANSITION SYSTEM

$\Gamma_E = \mathcal{E}$ $\quad T_E = \mathcal{N}$ $\quad o \in \{+, -, *, /\}$

$\rightarrow_E$ is defined by the following rules

$m + n \rightarrow_E p$ if $m + n = p$

the operation in the expression language

the operation as implemented in the underlying hardware/firmware/software

We need not assume that these operations are available in the underlying machine but then we would go into greater detail about numeral representation, how addition is performed in terms of the Boolean operations; ‘and’, ‘or’ and “not” (if you assume your numeral representation as purely binary). You could assume an underlying computational model which just consists of the Boolean operations and essentially give a minimal specification of how an addition algorithm is implemented using those Boolean operations. We can assume that the basic operations are already available in the machine.

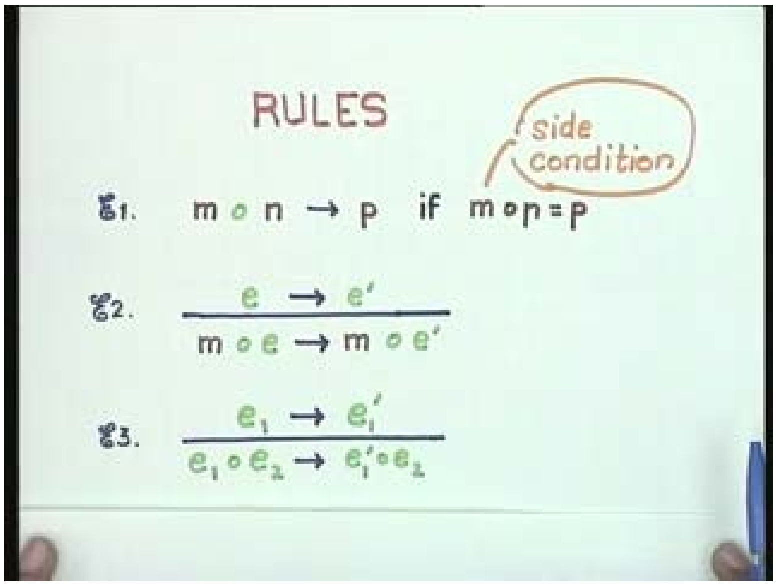
For example; I might make a statement $m + n \rightarrow_e p$.

This ‘+’ in the statement belongs to the language of expressions that we are interested in. The ‘+’ in m + n = p is the operation in the underlying hardware. So, it might either be implemented directly in hardware or firmware or software. The effect of performing the hardware operation on two numerals ‘m and n’ is some other numeral, p.

This transition, $m + n \rightarrow_e p$ acts as the basis for the specification of your transition system. You could go further and specify how each of the operations is to be implemented. As I said in a transition system method you can go into great detail about everything. You can also perform abstractions.

Here we are performing an abstraction with an assumption. The basis rule assumes that this operation is one of the four operations, which is already available in the machine. Note that we are looking upon this specification of a language as a form of symbol manipulation and that is how it can be operational.

[Refer Slide Time: 32:00]



RULES

side condition

$\mathcal{E}1.$ $m \circ n \rightarrow p$ if $m \circ n = p$

$$\mathcal{E}2. \quad \frac{e \rightarrow e'}{m \circ e \rightarrow m \circ e'}$$

$$\mathcal{E}3. \quad \frac{e_1 \rightarrow e_1'}{e_1 \circ e_2 \rightarrow e_1' \circ e_2}$$

I will assume that given two numerals m and n, if m binary operation n gives me a numeral p then in the expression language it also gives me a p. We are more interested in the high level language syntax of which $\text{m o n} \rightarrow \text{p}$ forms the basis.

If e can go to an expression e′ then m binary operation e can move to m binary operation e′. $\dfrac{e \rightarrow e'}{\text{m o e } \rightarrow \text{m o e}'}$

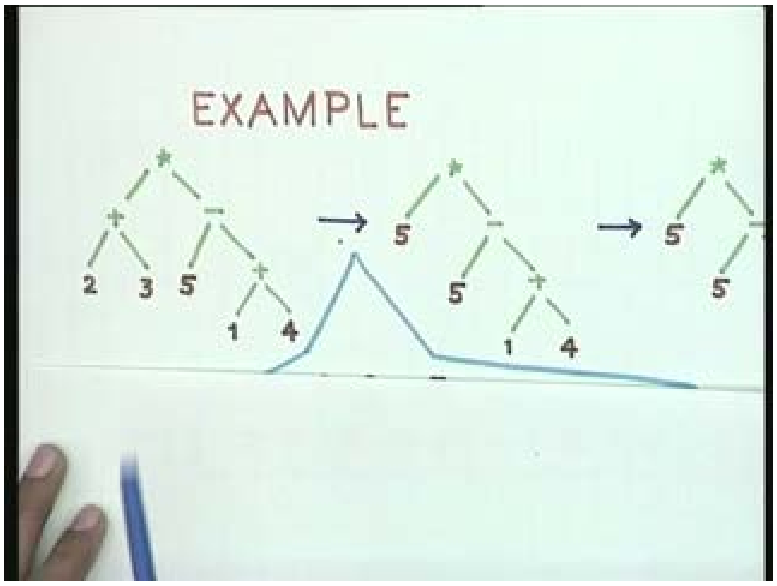
This is another very simple rule. If e_1 can go to an expression e_1' then e 1binary operation e 2 can go to e_1' binary operation 8.

$$\frac{\text{e}_1 \rightarrow \text{e}'}{e_1 o e_2 \rightarrow e_1' o e_2}$$

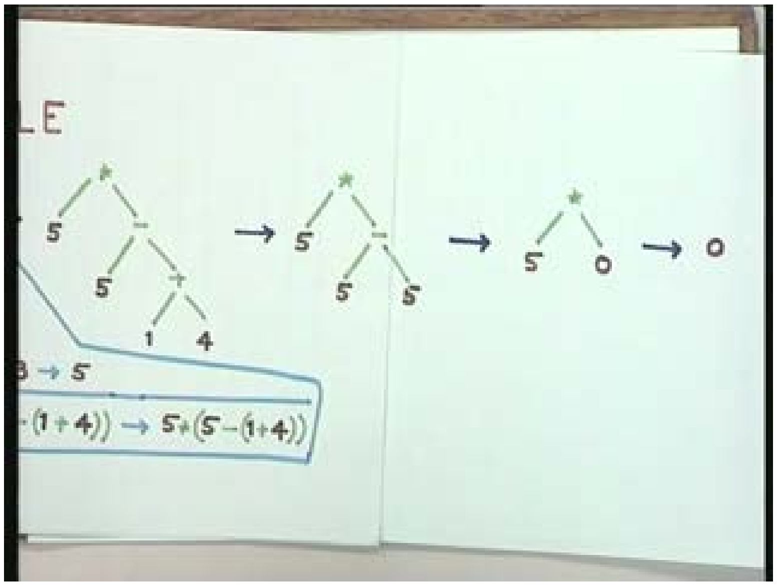
In fact these are all the rules that you require for this simple language. It is a succinct and clear specification but it contains a great deal of information.

Let us do a little example. Let us go back to what we said about expressions as a syntactic category. Expressions just denote values or an evaluation mechanism. There is no concept of store and there is no concept of environment. There are no complications associated with the expressions. The example that we take illustrates the concept of what transition system rules look like and what kinds of information you can derive from them. Take a tree from one location and move it by a purely tree manipulation process to another tree which is essentially that 2+3 has been evaluated and a sub tree has been replaced by a node. One tree is only replaced by another. The only change is that 1+4 has been replaced by 5. You go through the sequence of transitions.

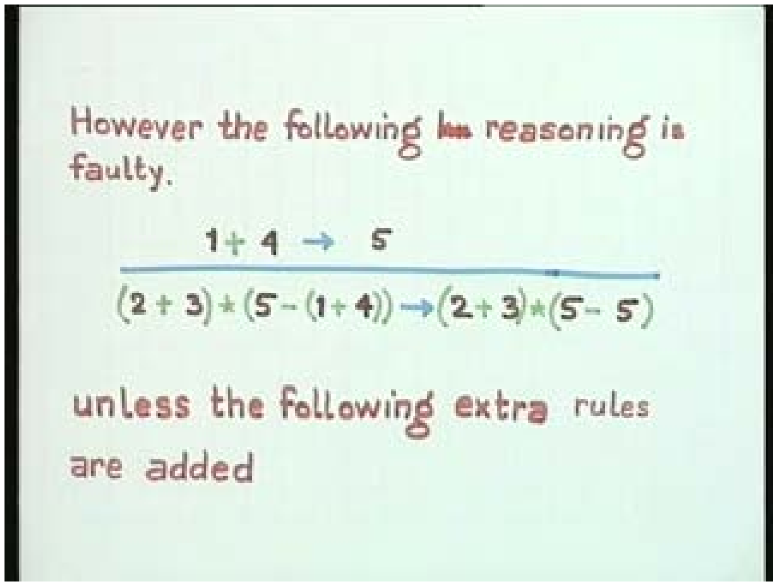
[Refer Slide Time: 34:10]



[Refer Slide Time: 34:13]



[Refer Slide Time: 35:08]



However the following ~~[illegible]~~ reasoning is faulty.

$$\frac{1+4 \rightarrow 5}{(2+3)*(5-(1+4)) \rightarrow (2+3)*(5-5)}$$

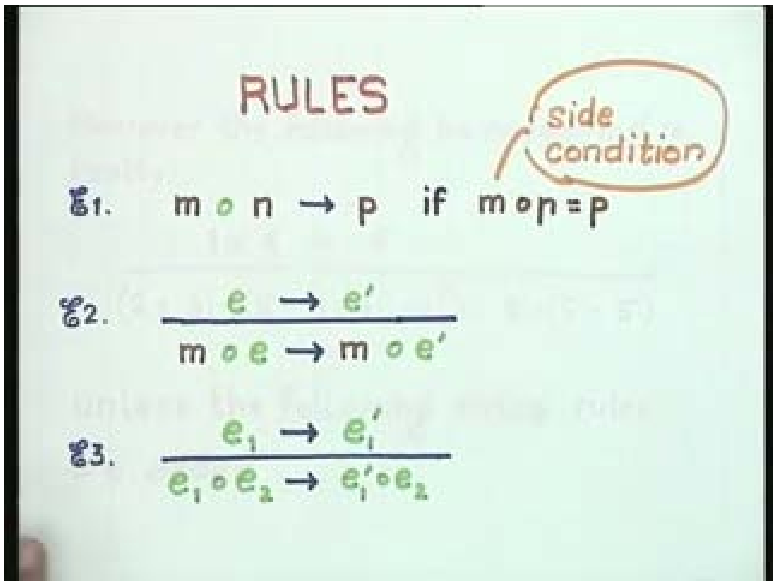
unless the following extra rules are added

The reasoning $\frac{1+4 \rightarrow 5}{(2+3)*(5-(1+4)) \rightarrow (2+3)*(5-5)}$ is absolutely faulty.

Is it just that a step is missing in between or is there something else? If you look at the rules, the rule e2 specifies that you can move a right hand side operand provided the left hand side operand is already a numeral. The rule e3 does not specify that you can take e_2 to e_2' and therefore you can infer e_1 binary operation e_2 goes to e_1 binary operation e_2'. Nowhere is it allowed. I have specified only positive rules so you cannot derive whatever is not allowed. For example; a rule like e6 is essential if you want to perform that kind of reasoning. The faulty reasoning can be rectified if you add rule e 6 which allows a right sub tree to perform a transition in a whole expression.

Supposing you allow rule e 6 and you add this extra rule which allows a right operand also to move. Given a complicated syntax tree there are a variety of possible transitions it can take and you no longer have deterministic behavior. You have non deterministic behavior.

[Refer Slide Time: 38:30]



RULES

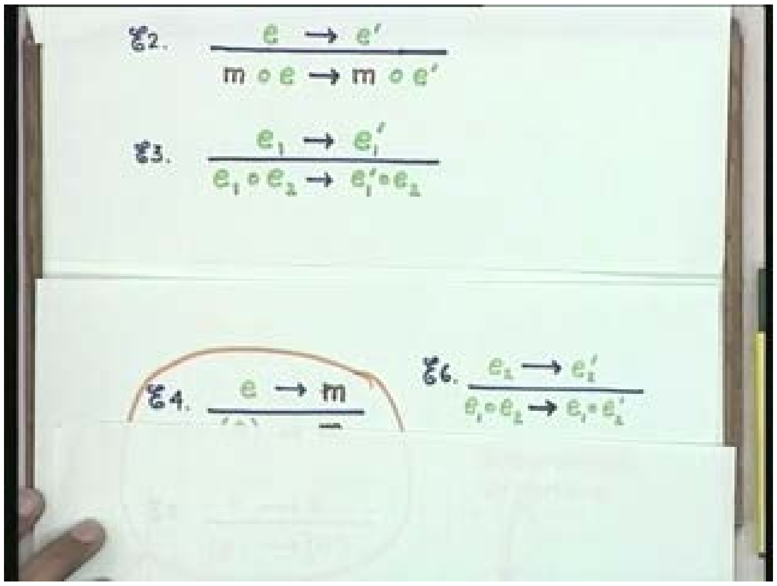
side condition

ξ1. $m \circ n \rightarrow p$ if $m \circ n = p$

ξ2. $$\frac{e \rightarrow e'}{m \circ e \rightarrow m \circ e'}$$

ξ3. $$\frac{e_1 \rightarrow e_1'}{e_1 \circ e_2 \rightarrow e_1' \circ e_2}$$

[Refer Slide Time: 39:35]



ξ2. $$\frac{e \rightarrow e'}{m \circ e \rightarrow m \circ e'}$$

ξ3. $$\frac{e_1 \rightarrow e_1'}{e_1 \circ e_2 \rightarrow e_1' \circ e_2}$$

ξ4. $e \rightarrow m$

ξ6. $$\frac{e_2 \rightarrow e_2'}{e_1 \circ e_2 \rightarrow e_1 \circ e_2'}$$

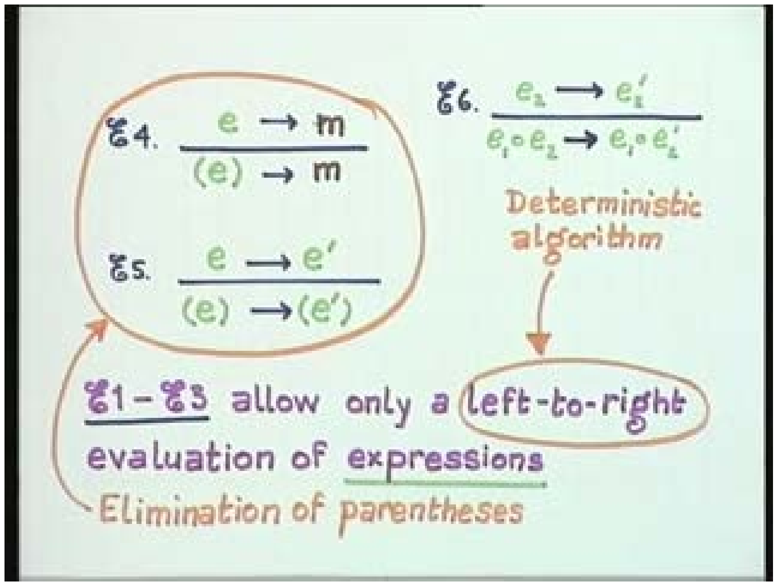
If you allow non deterministic behavior and it is an imperative language and you allow functions and side effects then it is not even clear that you are going to preserve any semantics. The results that you get from a program execution at that point will be completely dependent upon the order in which the runtime system chose to evaluate the expressions.

Addition is commutative but at least from a pragmatic point of view, it is necessary to clearly specify an order of evaluation at least in an expression language so that there is no confusion later when you expand the language out. I might take the PL0 and add more facilities or functions to it. Since you do not lose anything by specifying an explicit order of evaluation, it makes it convenient, pragmatic and decent. It does not give an implementer arbitrary choices or it does not produce conflict in interpretation. You could add further brackets and more rules but it is not necessary. Very often it just clutters up the matter at hand.

Determinism is very important from a pragmatic view point that is from the point of view of an implementer of the language. It is also very important for a user in case the language allows side effects. In fact that is the problem with many Pascal implementations. You will see in a turbo Pascal environment that you actually can get a choice of menus on how you want an expression to be evaluated and in what order you want expression to be evaluated but the manual itself is silent on that matter.

We should realize that especially in floating point arithmetic, for example, addition is no longer commutative and not even associative. Multiplication does not distribute over addition or subtraction. We will loose a whole amount of mathematical dogmas we were brought up with when we do actual floating point computation. Sometimes just for pragmatic reasons it becomes necessary to lift these constraints to the level of a language definition and at the level of a language definition you really do not lose anything by making it deterministic $\frac{e_1 \to e'}{e_1 o e_2 \to e_1' o e_2}$. It only helps the user and the language implementer to decide on a specific order.

[Refer Slide Time: 45:05]



$\mathcal{E}4.\quad \frac{e \to m}{(e) \to m}$

$\mathcal{E}5.\quad \frac{e \to e'}{(e) \to (e')}$

$\mathcal{E}6.\quad \frac{e_2 \to e_2'}{e_1 \circ e_2 \to e_1 \circ e_2'}$

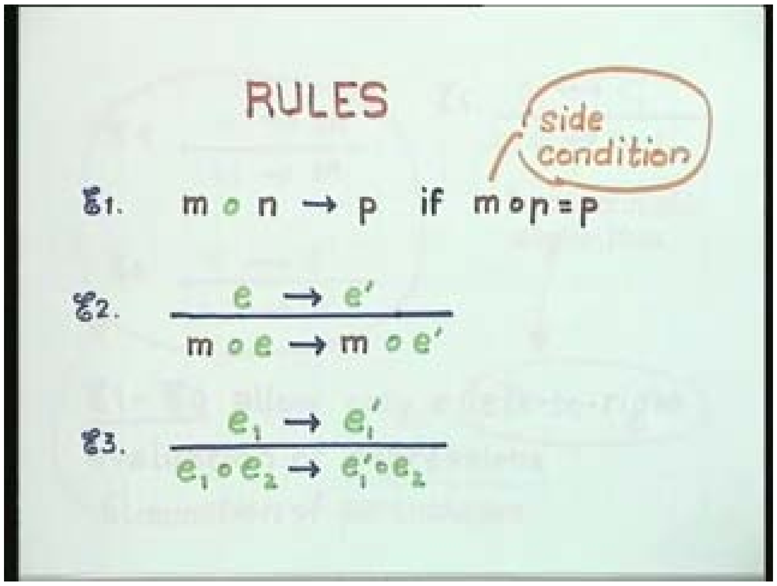
Deterministic algorithm

$\mathcal{E}1 - \mathcal{E}3$ allow only a left-to-right evaluation of expressions

Elimination of parentheses

Adding extra rules which allow non determinism may also be faulty sometimes. It is open to a variety of interpretations. In this case by having just three rules we have played absolutely safe. It allows only a left to right evaluation of the sub trees of the trees. Till you have reduced the leftmost sub tree to a single node you are not allowed to proceed with any of the right sub trees. That is what these three rules specify and they specify that by default. What you cannot do is being specified by default.

[Refer Slide Time: 45:40]



RULES

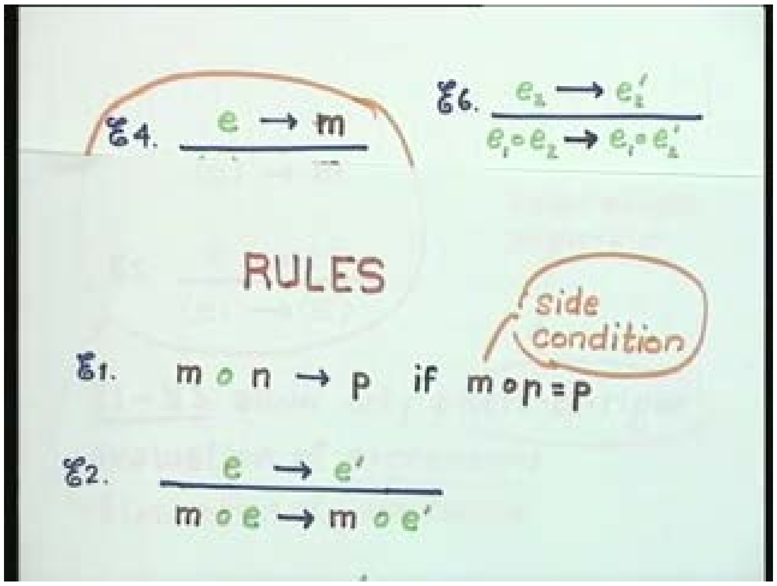
side condition

$\mathcal{E}1.$ $m \circ n \to p$ if $m \circ n = p$

$\mathcal{E}2.$ $\dfrac{e \to e'}{m \circ e \to m \circ e'}$

$\mathcal{E}3.$ $\dfrac{e_1 \to e_1'}{e_1 \circ e_2 \to e_1' \circ e_2}$

[Refer Slide Time: 46:18]



$\mathcal{E}4.$ $\dfrac{e \to m}{}$

$\mathcal{E}6.$ $\dfrac{e_2 \to e_2'}{e_1 \circ e_2 \to e_1 \circ e_2'}$

RULES

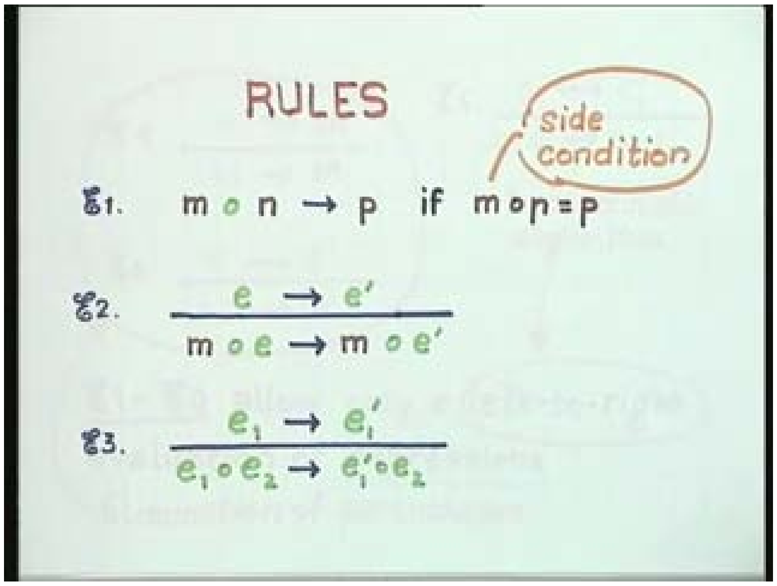
side condition

$\mathcal{E}1.$ $m \circ n \to p$ if $m \circ n = p$

$\mathcal{E}2.$ $\dfrac{e \to e'}{m \circ e \to m \circ e'}$

When you add a rule like e 6 you allow also simultaneous evaluations or independent evaluations of right sub trees; but adding such rules in order to presumably give more power to your transition system though well intentioned might actually lead to flaws in the semantics of the language and flaws in implementation. If your underlying computational assumption is that there are an unbounded number of possessors available to perform all these then allow this too. If your underlying implementation is such that you cannot have side effects then this is permitted. That is how a later feature added to a language can affect the complete semantics of a language. An expression language is actually trivial and simple, but the point is that the amount of information that you can derive from it has long term consequences on adding new features to the language which may not be obvious initially.

[Refer Slide Time: 47:46]



RULES

side condition

$\mathcal{E}1.$ $m \circ n \rightarrow p$ if $m \circ n = p$

$\mathcal{E}2.$ $$\frac{e \rightarrow e'}{m \circ e \rightarrow m \circ e'}$$

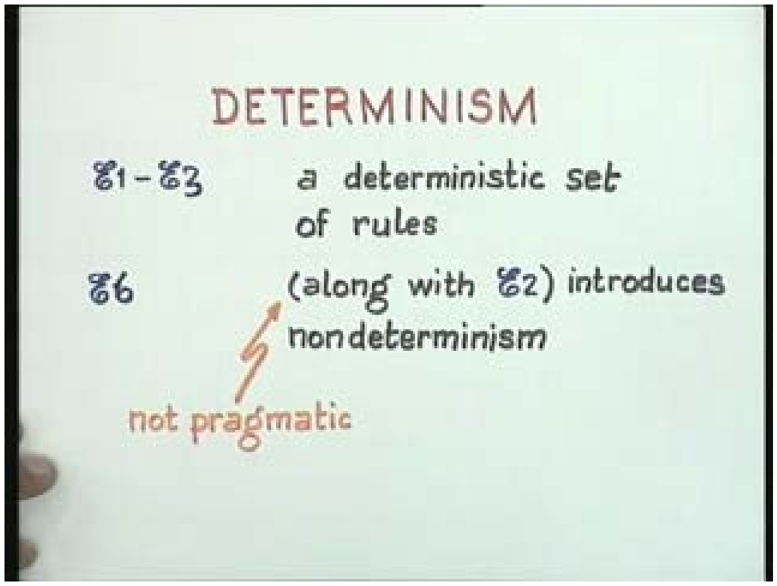
$\mathcal{E}3.$ $$\frac{e_1 \rightarrow e_1'}{e_1 \circ e_2 \rightarrow e_1' \circ e_2}$$

Actually I have taken a shortcut in the sense that I have used ‘o’ to represent one of four operators but as I said, since they are all part of the computational model and they are all readily available it does not really matter. You can look upon these three rules as really multiplied by four. There are twelve rules, one for each operator because ‘o’ is really not part of the language. It is a meta-variable for the four operators. We actually have twelve rules but it is concise, precise and it gives a deterministic order. So there is only one way a syntax tree can be reduced to a single node. So, you have to craft the semantics very carefully since everything that you cannot do is left unspecified. It has to be derivable from what you can do.

You can have a lot of trouble just crafting a few rules. It is not necessary to have a minimal set of rules. Just to play safe you might add more rules but then the consequences of that should be first realized. Does it allow a kind of non determinism?
Do we want that non determinism? We might want it if we are designing the semantics of the expression language as part of an implementation for a machine like the Parham with 64 processors or some hyper cube.

It might be better to have that non determinism in the semantics and get rid off side effects in function evaluation when I add functions to the language. We might explicitly ban side effects in the larger interest of program clarity. But whatever decision I take, I have to be absolutely clear which is that if there is some non determinism do I want it? In a multiprocessor environment it makes a lot of sense to introduce these extra rules which allow left to right mixed with right to left etc. If your pragmatics is such that whatever you do for the multiprocessor machine should also be applicable for a uniprocessor machine in case you wanted to port the language for long term use of that kind, then you must think in terms of what are the constraints. You would want to know if you want that non determinism. Will having that non determinism actually create more problems or will it solve more problems? It could work either way.

[Refer Slide Time: 52:00]



DETERMINISM

$\mathcal{E}1 - \mathcal{E}3$ a deterministic set of rules

$\mathcal{E}6$ (along with $\mathcal{E}2$) introduces nondeterminism

not pragmatic

Very often, we will just take an implicit assumption that we want everything to be also implemented for a simple uniprocessor but nowhere in the rules are they actually specified. In that sense you are totally abstracted away from the machine except for the basic underlying computational model. It requires a really simple transition system which seems so trivial to point out the important aspects of the difference between determinism and non determinism, the difference between the desirability and the undesirability of the two and the difference of how you should craft rules so that what you desire most is made specific in the rules and what you do not desire is also somehow derivable from the rules. So, writing semantics is difficult even for trivial cases, which is one of the reasons why it takes so much time even to write a few rules. We will continue with declarations and commands of PL0 in the next lecture.