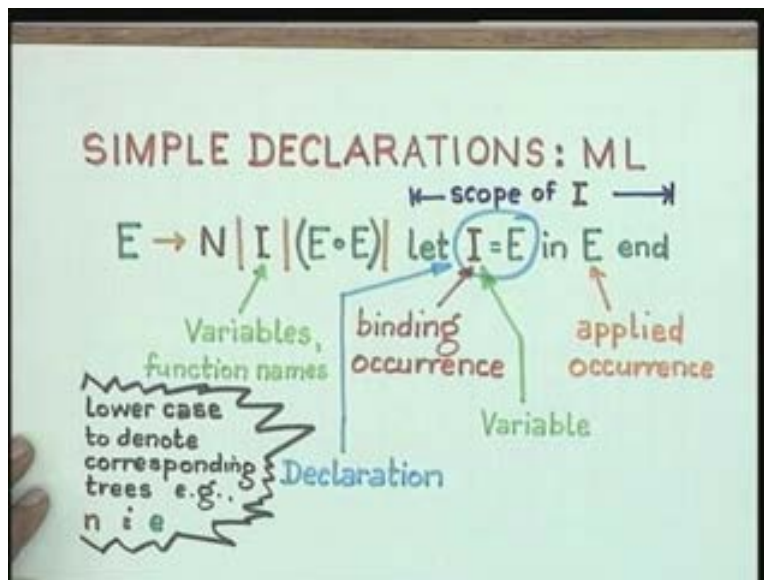


**Principles of Programming Languages**  
**Prof: S. Arun Kumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology**  
**Delhi**  
**Lecture no 12**  
**Lecture Title: Declarations**

Welcome to lecture 12. We will continue what we did last time with an emphasis on declarations. Before that I will quickly go through the transition system as we defined last time. We were looking at simple ML like declarations. We have variables and for the present we will just assume that variables have only values and they are not functions.

(Refer Slide Time: 01:30)

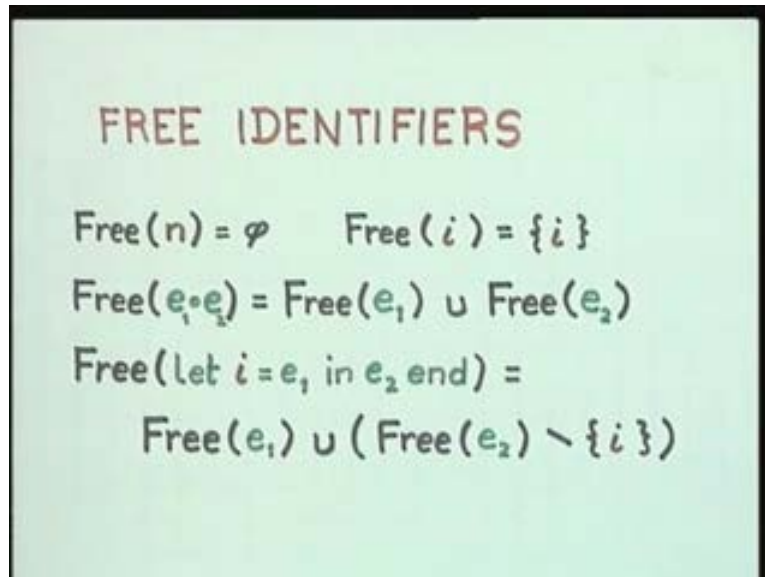


We have the usual binary operators and a let construct 'Let I = E in E end'. This I = E is the binding occurrence of the let construct. A binding occurrence in some books is also called a defining occurrence. The E of 'E end' is an applied occurrence. Declarations are also called definitions but we will call them declarations.

This  $E \rightarrow N \mid I \mid (E \circ E)$  is the grammar with non terminals.

I will use corresponding low case letters; 'n, i, e' to distinguish the fact that I am actually interested in syntax trees and not in the grammar per say which has other implications besides that for semantics.

(Refer Slide Time: 02:05)

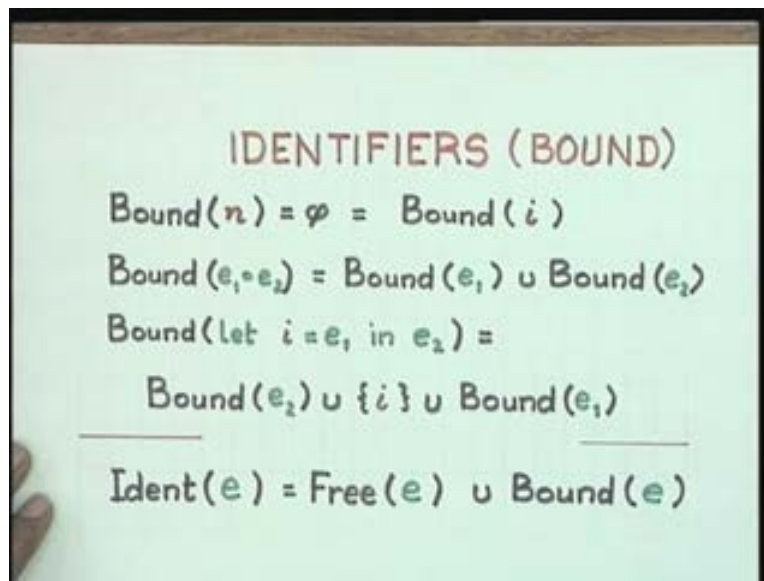


FREE IDENTIFIERS

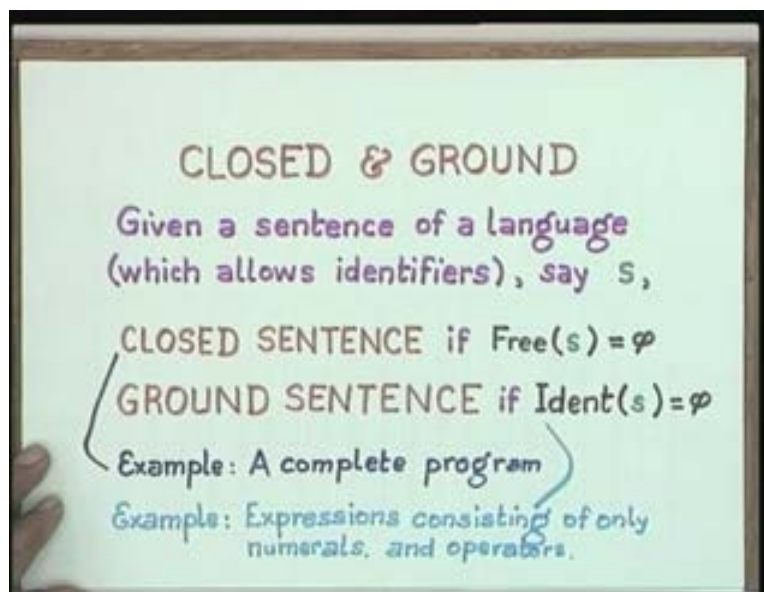
$$\begin{aligned}\text{Free}(n) &= \varnothing & \text{Free}(i) &= \{i\} \\ \text{Free}(e_1 \cdot e_2) &= \text{Free}(e_1) \cup \text{Free}(e_2) \\ \text{Free}(\text{let } i = e_1 \text{ in } e_2 \text{ end}) &= \\ &\quad \text{Free}(e_1) \cup (\text{Free}(e_2) \setminus \{i\})\end{aligned}$$

We defined the notion of a free identifier in an expression and in particular in a let-construct 'Let I = E in E end' the identifier that is declared is not free. We defined bound identifiers and the set of all identifiers is just the set of all free and the set of all bound identifiers.

(Refer Slide Time: 02:20)



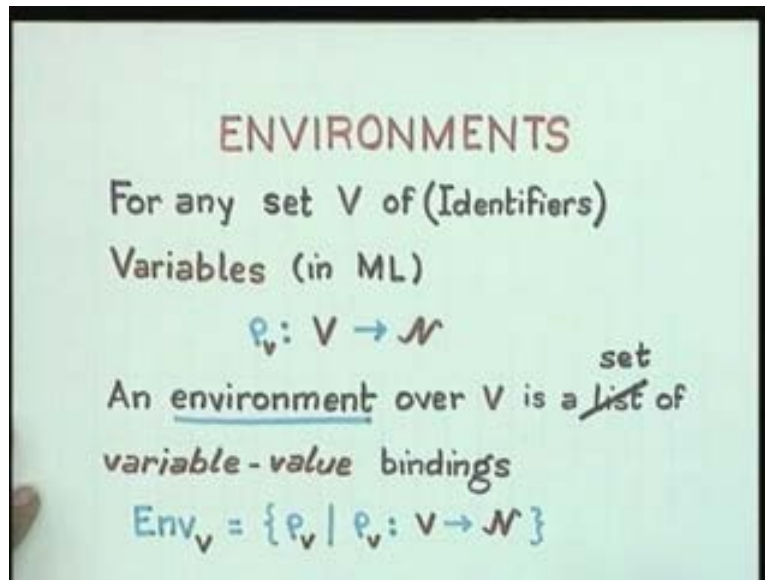
(Refer Slide Time: 02:35)



We defined the notion of closed terms of a language and ground terms of a language. A complete program is closed and a term without variables is ground. Then we defined the notion of an environment as a function. An environment over a set  $V$  of variables is just a function which maps variables onto a particular domain of values and an environment is a

set of variable-value bindings and the function  $P_v : V \rightarrow \mathcal{N}$  actually defines the binding. We could go a little further and consider the set of all possible environments.

(Refer Slide Time: 03:05)



I will define list environment as the set of all possible environments over the set of all possible variables. If I consider unions rather than what is known as a disjoint summation then there is a possibility that at some point you might have two environments which are supposed to be distinct in some logical way but it so happens that they represent the same function.

Supposing you have two sets  $A$  and  $B$  then a disjoint union of the two sets  $A$  and  $B$  is equivalent to having an identity associated with each element in the two sets. For example; if the two sets were not disjoint and if they had some common elements then  $A \cup B$  will just consist of all the elements of  $A$  and  $B$  without actually specifying an identity as to where that element came from. Let us take some standard representations. If I take the union of the set of all natural numbers and the set of all role numbers of B.Tech students then that is just going to be the set of all natural numbers because all your role numbers are natural numbers. But if I take a disjoint union of the set of all natural numbers and the set of all role numbers then there is a distinct identity associated with

each element. A role number like 94141 will occur twice in the set but once as a natural number and once as a role number. The best way to think of disjoint unions is as having tagged the identities of the elements individually.

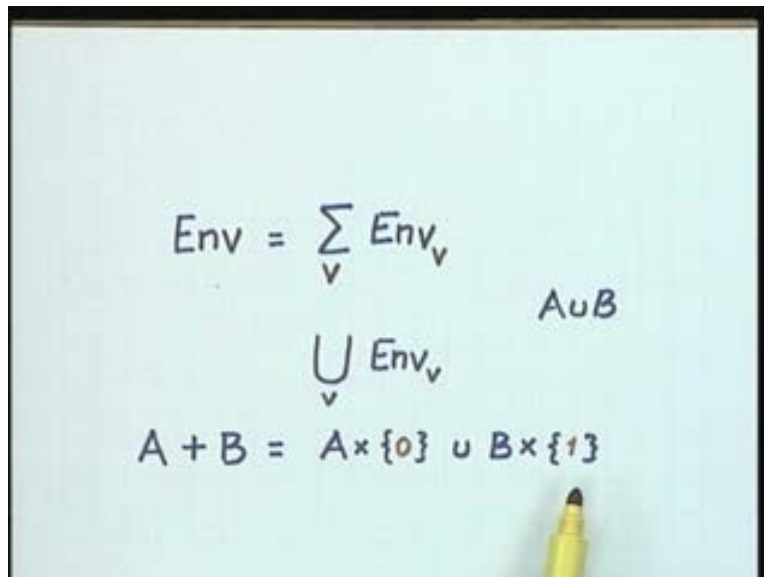
If you have these two sets then I can look upon the disjoint union as;

$$A + B = A \times \{0\} \cup B \times \{1\}$$

Similarly, if I take the set of the disjoint union of the set of all role numbers of B.Tech students then by ' $A + B = A \times \{0\} \cup B \times \{1\}$ ' I am actually tagging them appropriately.

I would actually have 94141, 0 as one element and 94141, 1 as another element and they will be distinct and the tag of 0 or 1 specifies which parent set it actually came from and you actually encounter disjoint unions in programming.

(Refer Slide Time: 07:55)


$$\begin{aligned} \text{Env} &= \sum_v \text{Env}_v \\ &\quad \bigcup_v \text{Env}_v \quad A \cup B \\ A + B &= A \times \{0\} \cup B \times \{1\} \end{aligned}$$

The variant record construct in PASCAL for example, specifies a disjoint union and in the case of a tag you get some values according to the tag. A variant record in PASCAL like language actually specifies a disjoint union of two sets where the tag, the variant part of the record, actually disambiguates whether an element is from one or another.

The variant construct for example, works as follows.

*type*

R = record

Case tag: (0, 1) of 0: a: integer

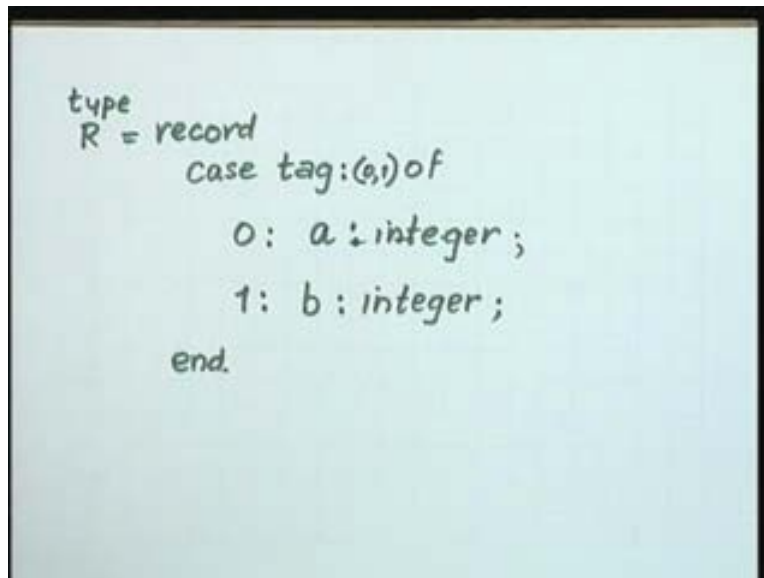
0: a: integer;

1: b: integer;

end

I could have case 0 of; I could have single variable a, which is integer and in that case I could have a single variable b which is also of type integer end case. This kind of declaration in PASCAL really specifies that I am using integers but I am using integers to mean two different things and what I mean by them are specified by the tag.

(Refer Slide Time: 10:23)



```
type
R = record
  case tag:(0,1) of
    0: a: integer;
    1: b: integer;
  end
```

Disjoint unions of data types are standard constructions. Although they are not used very much in mathematics, they are definitely used in programming.

Just in order not to confuse matters the environment  $Env = \sum_v Env_v$  is a disjoint union and not a mere union. It is a disjoint union or a set of all possible environments that you can have.

(Refer Slide Time: 10:55)

$$Env = \sum_v Env_v$$

$$\cancel{\cup_v Env_v} \quad A \cup B$$

$$A + B = A \times \{0\} \cup B \times \{1\}$$

A separate grammatical entity has a separate logical meaning. We looked at sequential declarations, nested declarations and we defined the notion of updation of an environment. Disjoint union becomes effective when you have two distinct environments on variable sets which are not necessarily disjoint and you want to give a preference as for any identifier what value it should take in the updated environment.

The binding in this updated environment  $P(i)$  of any identifier 'i' is  $P_2(i)$  if  $i \in V_2$ .

The 'i' could belong to  $V_1$  also but that does not matter and if 'i' does not belong to  $V_2$  then it is whatever is defined by row 1,  $P_1(i)$ .

(Refer Slide Time: 12:15)

ENVIRONMENTS (Contd.)

Variable sets.  $V_1, V_2, V = V_1 \cup V_2$

$P_1 : V_1$  i.e.  $P_1 \in \text{Env}_{V_1}$

$P_2 : V_2$  i.e.  $P_2 \in \text{Env}_{V_2}$

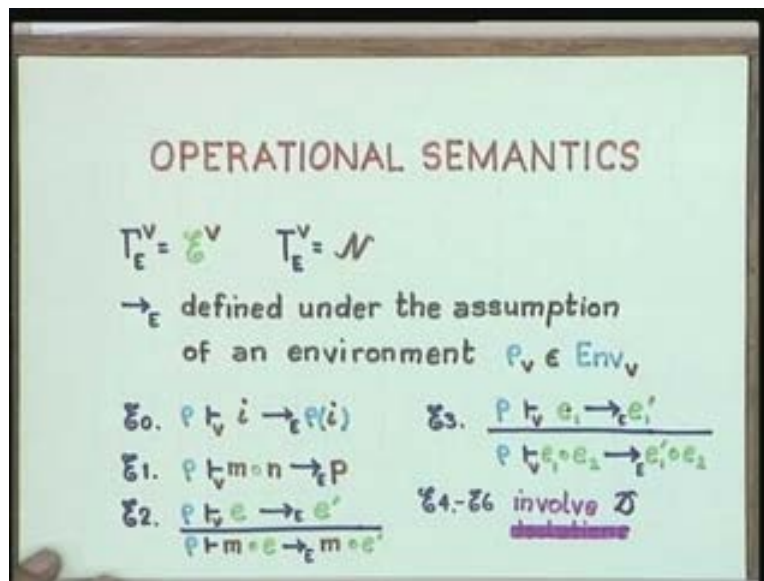
$P = P_1[P_2] : V$  i.e.  $P \in \text{Env}_{V_1 \cup V_2}$

$$P(i) = \begin{cases} P_2(i) & \text{if } i \in V_2 \\ P_1(i) & \text{if } i \in V_1 \setminus V_2 \end{cases}$$

We looked at the operational semantics of expressions with a usual notion of configurations except that we have to add the set of variables that we are actually interested in. The meaning of an expression is not clear unless you can assign a value to an identifier and that assignment of a value to an identifier is performed by the environment and so, you have the three rules with an extra assumption. You can assume that ' $P$ ' acts as an extra assumption. Under the assumption of an environment row the meaning of ' $i$ ' is whatever is the binding defined by the row on ' $i$ '. Similarly, the meanings of all the expressions depend on the assumption of the environment.



(Refer Slide Time: 13:26)



Then we looked at the semantics of the 'let construct'. Given any let expression  $\text{let } d \text{ in } e \rightarrow \text{let } d' \text{ in } e$  provided  $d \rightarrow_D d'$  and  $d$  is a declaration. So, there is a separate transition system for declarations. That is why there is a subscript of  $d$  whereas the entire 'let construct' defines an expression. The transition  $E$  is in the expression transition system. In order to evaluate a 'let construct' you are depending upon first evaluating the declarations that the 'let construct' contains and those declarations could be complex so you require the rule  $d \rightarrow_D d'$  and all our transition systems are symbol manipulations.

This declaration  $d \rightarrow_D d'$  might during the process of evaluation, get transformed into some other declaration  $d'$  and eventually after several applications, presumably you would have identified an environment row prime,  $P'$ .

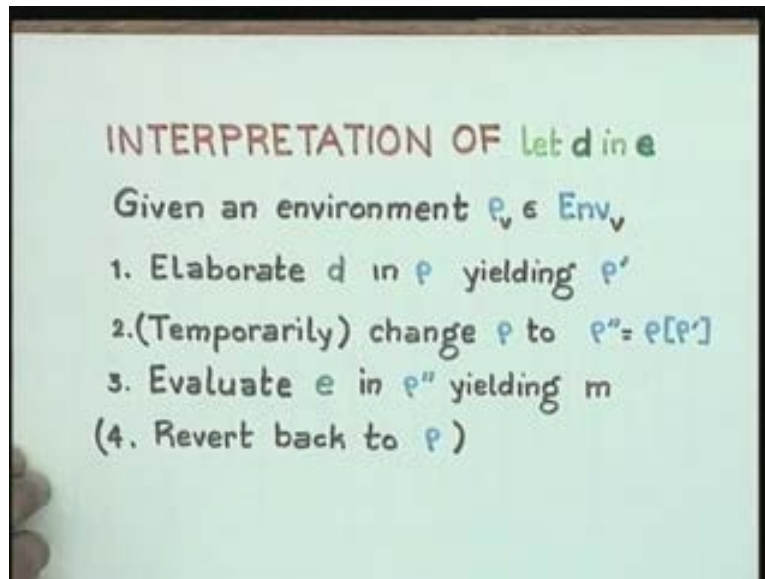
Declarations give you environments and you would probably get (an environment row prime)  $P'$  and now the expression 'Let  $P'$  in  $e$ ' is strictly speaking not in the syntax of the language. It is an expression which we are using in order to specify meanings in a systematic fashion. We have to specify one step transitions. So, there is an intermediate state in which the environment has been created but 'Let  $P'$  in  $e$ ' is some sort of an intermediate specification rather than being part of the syntax.

In the environment the meaning of the expression  $e$  can transit to the expression 'Let  $P'$  in  $e'$ ' with the  $e'$  provided in the updated environment  $P[P']$  you can show that  $e \rightarrow e'$ . Several applications of the rule script  $e5$  with the rules for ordinary expressions should finally yield a constant and then the meaning of the expression 'Let  $P'$  in  $m$ ' is just the constant ' $m$ '. However, it is necessary to know that we have intermediate steps and eventually we are still interested in the original environment row. We are looking at the evaluation of an expression which contains a declaration in the original environment row and in the process of evaluating, you create a temporary change in environment in order to enable the evaluation of the expression ' $e$ ' and once the expression has been evaluated you revert to the original environment,  $P$ . The original environment is all that you have.

The temporary changes are reversible changes and once you look at the data structures in pragmatics, you will see that they really involve no work at all. An interpretation of the 'let construct' according to our rules can be loosely specified in the following way. Given an environment row ' $P_v \in Env_v$ ' you first elaborate the definition  $d$  or the declaration  $d$  in row,  $P$  and the elaboration of a declaration yields an environment row prime,  $P'$ . So, you temporarily change row to row updated with row prime,  $P$  to  $P'' = P[P']$  and evaluate  $e$  in the new environment,  $P''$  till it yields an  $m$  and then you revert to the original environment,  $P$ .

Several applications of the expression transition system should essentially boil down to the steps that are specified. I have specified them as a sort of an algorithm but nowhere in our rules have we actually ever decided on a data structure or an algorithm. We just have to apply the rules very much like we did in the towers of Hanoi problem whether in the specifications of the towers of Hanoi problem or in the execution of the towers of Hanoi problem. We just apply rules which are not really algorithms but they actually hide algorithms inside them and some of the assumptions actually hide data structures which are fairly natural inside them and the whole set of rules is precise, unambiguous and framed by induction on the structure of the syntax.

(Refer Slide Time: 19:21)



Let us now look at that semantics of declarations. The rule e4: 
$$\frac{P \text{ to } d \rightarrow_D d'}{P t_v \text{ Let } d \text{ in } e \rightarrow \text{Let } d' \text{ in } e}$$

obviously involves some rules that we still do not know.

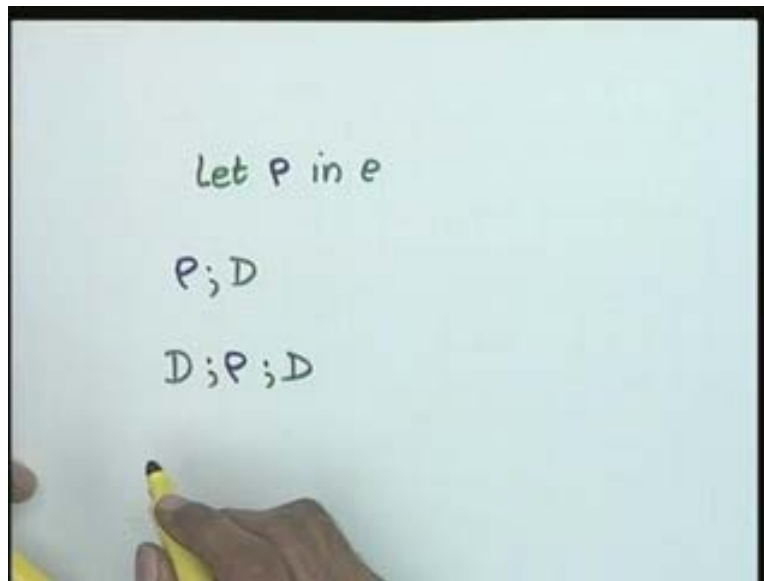
You cannot conclude  $P t_v \text{ Let } d \text{ in } e \rightarrow \text{Let } d' \text{ in } e$  unless you can do something about the declaration transitions  $P \text{ to } d \rightarrow_D d'$ . That means that we have to define a new set of configurations for declarations and define a new set of rules to process declarations also in a syntax directed fashion.

In the expression language, 
$$\begin{array}{l} E \rightarrow N \mid I \mid (E \circ E) \mid \text{Let } D \text{ in } E \\ D \rightarrow I = E \mid D ; D \end{array}$$
 I have added a row,  $|P$ . You can

think of the grammar  $D \rightarrow I = E \mid D ; D$  as being extended by another production which from a declaration gives you the row  $|P$ . The row  $|P$  is strictly not part of the language but part of our semantical specification language. It is not part of the original language and if you add the production  $|P$  to the language then you have various possibilities. For example; the moment you add the row to the language you have possibilities like 'Let  $P$  in  $e$ '.

You have other possibilities for declarations. You can have declarations of the form  $P; D$ . You can also have  $D; P; D$ . You have all the extra constructions possible when you add the row,  $P$  as a new production.

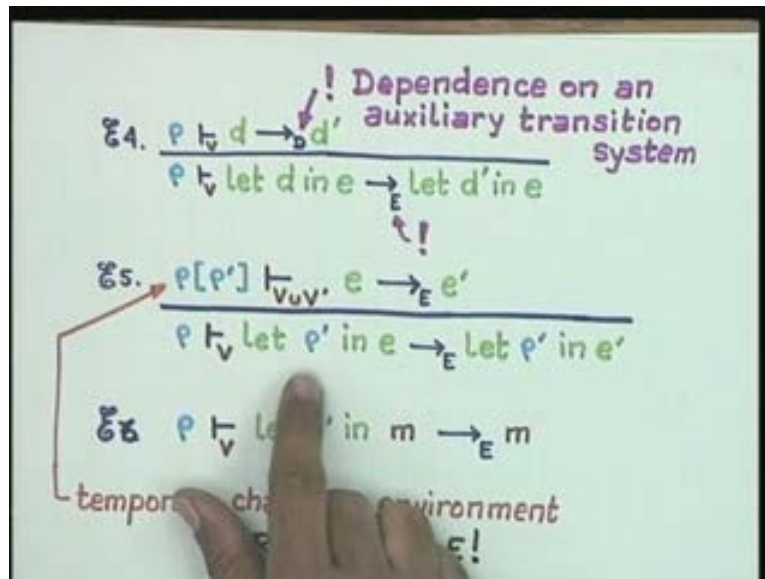
(Refer Slide Time: 22:10)



We require the extra row as part of our semantical specification and not really as part of the language. We have used some of our rules which had to deal with expressions.

For example; that is what we have used in ' $\text{Let } P' \text{ in } e \rightarrow_E \text{Let } P' \text{ in } e'$ '. We have used the construction where you can have an environment  $P'$  itself as being part of the declaration but not part of the language. It is not part of the original language or specification language. It is part of the notation that we are using in order to specify the meanings of constructs.

(Refer Slide Time: 22:55)



The set of configurations that we are interested in for declarations is given by the set  $D$  in ' $\Gamma_D^V = D^V$ ' where  $D$  is not just of the language constructs but also includes that extra production  $D$  goes to row. We had the base case of this declaration  $D$  as  $I = E$  ( $D \rightarrow I = E$ ) and sequencing of declarations. Now you can assume that for our specification purposes our base cases include not just  $I = E$  but also  $|P$  and you can therefore mix actual syntactic declarations with environments according to the particular grammar in anyway.

You have a collection of configurations which goes beyond just the syntactic declarations.  $D$  in ' $D^V$ ' should be understood as specifying all the possible declarations that are specified by the extended grammar with the production row  $|P$ . The set of terminal configurations is just the set of environments ' $T_D^V = Env_v$ '.

Note that our terminal configurations the set  $T_D$  has to be a subset of  $\Gamma_D$ .

That is also a good reason why you should include the row as part of the semantic specification language  $T_D \subseteq \Gamma_D$ . If you did not include the row then this important condition  $T_D \subseteq \Gamma_D$  will not be satisfied.

As far as just our semantical specification is concerned we will allow all these conditions. It is a patch work way of making our language of meanings more expressive. In general the language of meanings has to be more expressive than the syntactical language. That is how you get specification languages whose particular cases might be implementations in an existing language.

I already mentioned once that in general, specification languages might contain constructs that probably cannot even be implemented. In that sense in order to specify meanings you require a language that is more expressive but it is not really a formal language; it is just a language loosely specified in mathematics. I could have specified  $\Gamma_D^V = D^V$  such that it satisfies the condition  $T_D \subseteq \Gamma_D$  by including a whole lot of conditions which exactly specify the set of syntactic declarations and the mixtures of syntactic declarations and environments that are allowable in  $\Gamma_D^V = D^V$ .

But this 
$$\begin{array}{l} E \rightarrow N | I | (E \circ E) | \text{Let } D \text{ in } E \\ D \rightarrow I = E | D; D \end{array}$$
 seems a very simple and easy solution to the problem of how you can mix semantical elements with syntactical ones in order to specify your semantics.

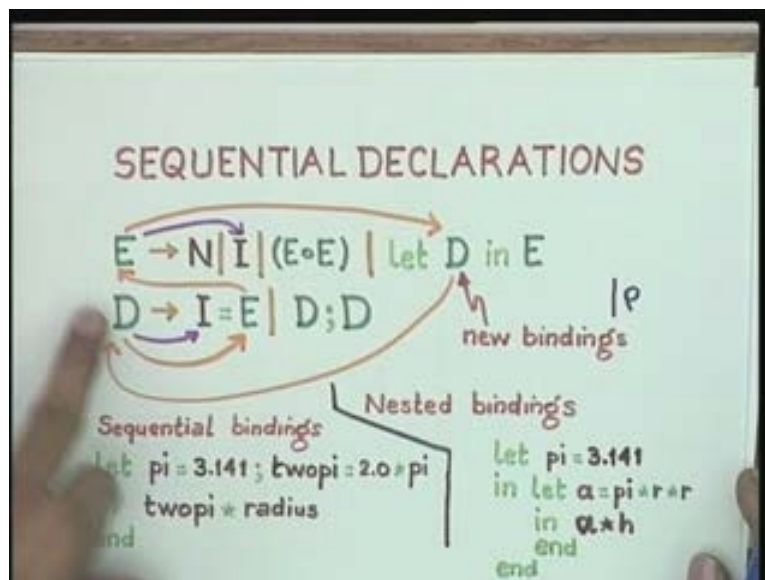
The set of configurations  $\Gamma_D^V = D^V$  includes all those mixtures of the abstract environment row because they signify intermediate steps in your computation. One way of specifying the intermediate steps of your computation is to actually write out the data structures completely and write out the algorithms completely. But that becomes too dependent on a particular implementation. If you are giving a language definition you would like to abstract away from particular implementations and make it as general as possible and you need a very simple and neat way of making your specification more expressive so that you can specify lower level details without compromising on the abstraction from particular implementations.

Now let us look at declarations. The base case of the declarations indicates that we are looking at trees. A single node tree is still a tree and not a non terminal in the sense of the

syntax of the language. The effect of a syntactical construct that represents a tree is to just give you a value binding. You do not really require an environment row in order to create a new environment,  $\{i = m\}$  but that is beside the point because you will require (the row)  $P$  anyway as you apply the rules.

What if a construct is a full bodied expression in itself? If a construct is a full bodied expression then you have to evaluate the expression in the environment row. If we already know the evaluation of expressions in a particular environment then the rule will be applied several times till the expression reduces to some constant at which time you can apply the prior rule to create the new environment. Such a construct is like the induction step which again depends upon the transition system for expressions.

(Refer Slide Time: 30:15)



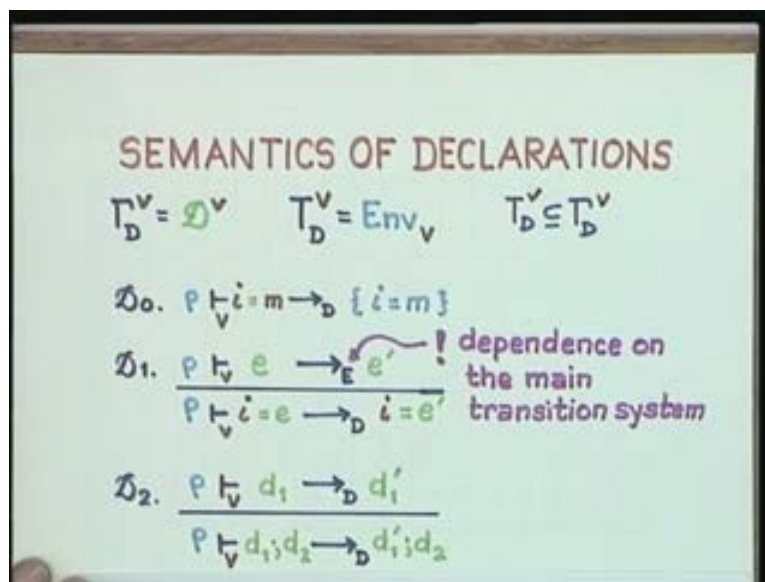
I have shown you dependencies in our sequential

declarations  $E \rightarrow N \mid I \mid (E \circ E) \mid \text{let } D \text{ in } E$  and  $D \rightarrow I = E \mid D ; D$ . The fact that there is an  $E$  means that you get

the dependencies circularly but of course it is not a circular definition because if you do the induction on the syntax, which is finite, the induction on the syntax shows that you have a descending chain of complexity of expressions.

The induction is perfect but it is just that it looks circular. Supposing you have complex declarations of the form  $d1; d2$  then it will move to some  $d1'; d2$  provided in an environment row  $d1$  can move to  $d1'$ . The movement of a declaration from  $d1$  to  $d1'$  is closely linked to the application of the rule  $d1$  where only the expression moves. If you assume that  $d1$  is just a single declaration of the form  $X = \text{some expression}$  then the movement of  $d1 \rightarrow d1'$  is linked to the movement of  $e$  to  $e'$ .

(Refer Slide Time: 31:45)



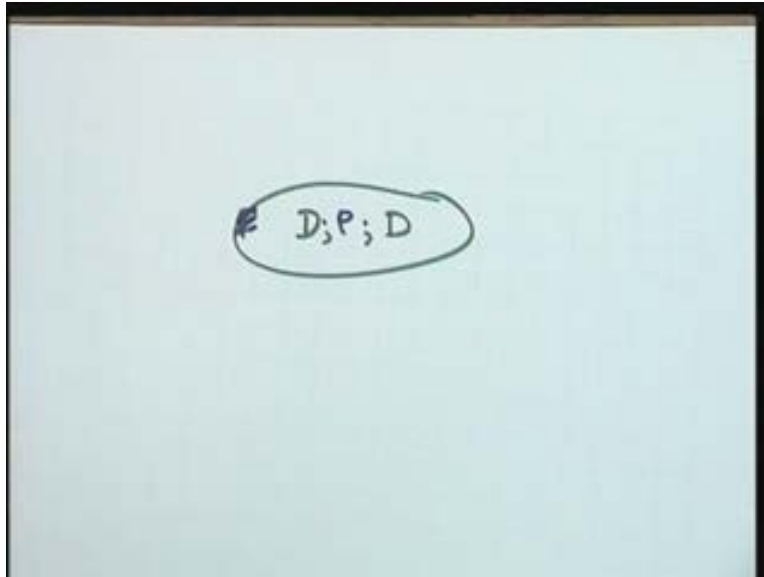
Again as we did in the case of expressions, we are looking at left to right sequential evaluations. This means in particular that even though in the domain of configurations we have allowed configurations of forms  $D; P; D$  in the set  $\Gamma D$  if you strictly follow a left to right evaluation of declarations you will not obtain such kinds of configurations ever.

But that does not matter because in any mathematical specification you might always have redundancy. It implies that according to our rules such configurations  $D; P; D$  will never move anywhere, where the left hand  $D$  is a purely syntactical entity and so is the right hand  $D$ .



Such configurations will never have any transitions. They will be what are known as stuck configurations.

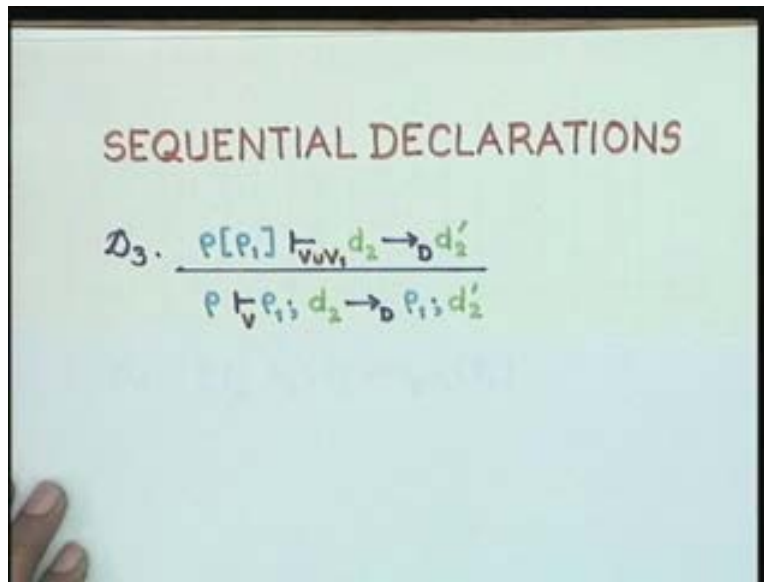
(Refer Slide Time: 33:20)



Lastly, we have to know what the effect of sequencing is on declarations. Supposing you have already evaluated a declaration and produced an environment  $P_1$  and then the evaluation is sequenced in such a fashion where say  $d_2$  cannot move to  $d_2'$  unless in the modified environment with the new set of variables,  $d_2$  can move to  $d_2'$ . In order to evaluate sequences of declarations you should have first provided a new environment.

In a more complex setting where you are not strictly constrained by left to right evaluations where you might allow non deterministic transitions, you could add extra rules such that the configurations also become meaningful. This aspect is there in all programming languages.

(Refer Slide Time: 34:35)



Supposing you have a sequence of constant declarations; the second constant declaration could depend on the first constant declaration. You could for example give a complex set of rules which specify that given three declarations of the form  $d_1; d_2; d_3$ , if you are perfectly clear that there are no free variables in  $d_2$  which occur in  $d_1$ , there is absolutely no reason why you should not evaluate  $d_1$  and  $d_2$  independently. There is absolutely no reason then why you should not evaluate  $d_2$  first and produce the  $D; P; D$  environment row and then start evaluating  $d_1$  and  $d_3$ . But as I said we will make it clear that our rules are such that you can have a uniprocessor implementation.

The interpretation of  $d_1; d_2$  is just that you have a given environment row and then you elaborate  $d_1$  in  $P$  to yield a new environment  $P_1$ . Then you elaborate  $d_2$  in row updated with  $P_1$  yielding  $P_2$  and then the result of elaborating  $d_1; d_2$  in  $P$  is just  $P_1 [P_2]$ .

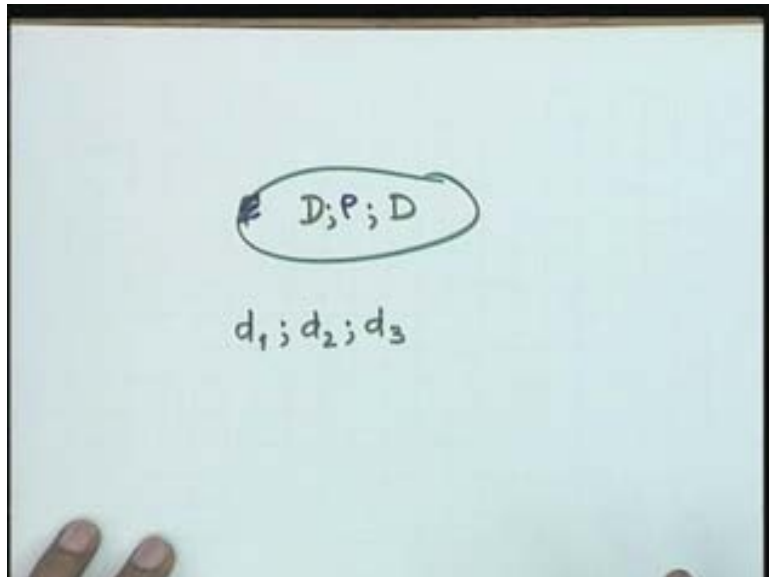
The interpretation does not preclude certain possibilities which are not allowed in most languages. But for example they are allowed in ML. Take an ML like program of the form:

```

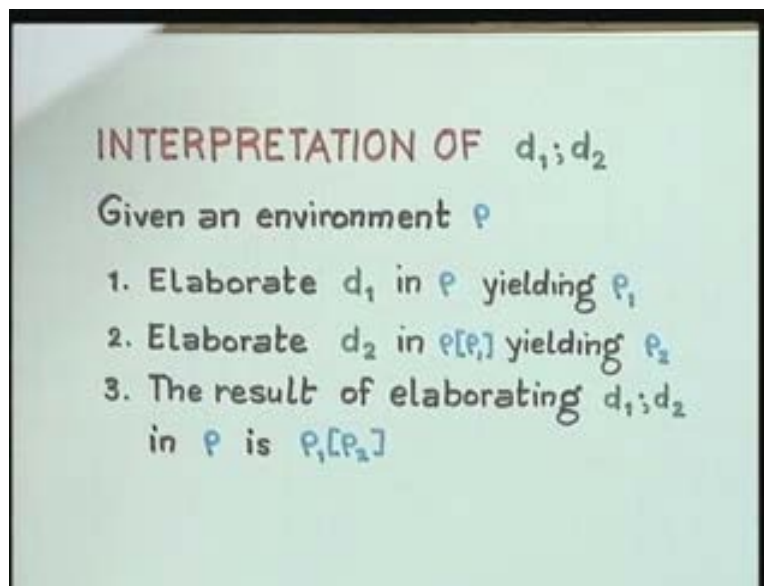
Let  $x = 3; x = x + 1$ 
in  $x^*x$ 
end

```

(Refer Slide Time: 36:02)



(Refer Slide Time: 36:55)



It is not allowed in a PASCAL constant declaration. You cannot have the same identifier occurring in the same sequence of constant declarations twice on the left hand side. But

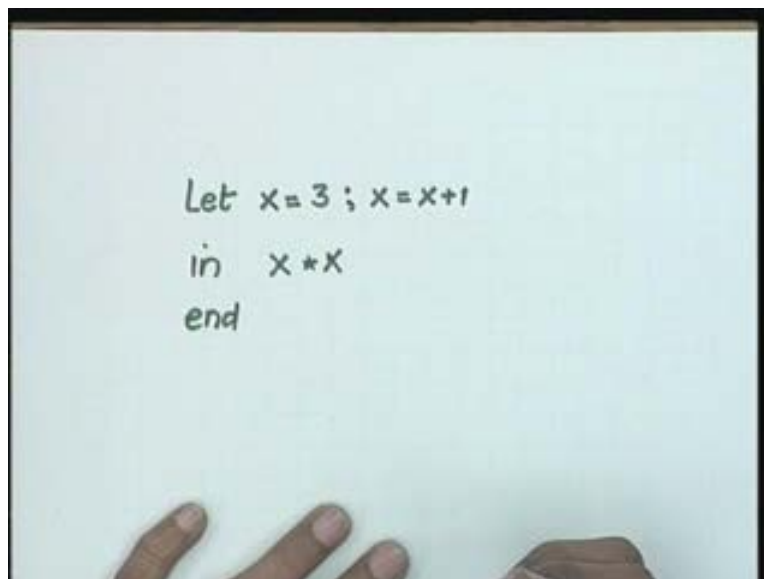
in the way we have defined the declaration, it really allows you to first declare it and use it. So, the updated environment will have  $x = 3$  after you have processed the declaration and that  $x = 3$  will be used in  $x = x + 1$  to give you a new updation of  $x$  which is  $x = 4$  and in that updated environment  $x * x$  will be evaluated and you are going to get 16.

Let us just go through a small example. Here is an example of a nested 'let construct'.

You have `let x = 3 in let y = x + 2; x = y + 1 in x + y` and you have various syntactical elements in it which I have named `e0 e1 e2 e3 e4`.

It is a complete ML program in itself and I would like to evaluate it in an empty environment.

(Refer Slide Time: 38:45)

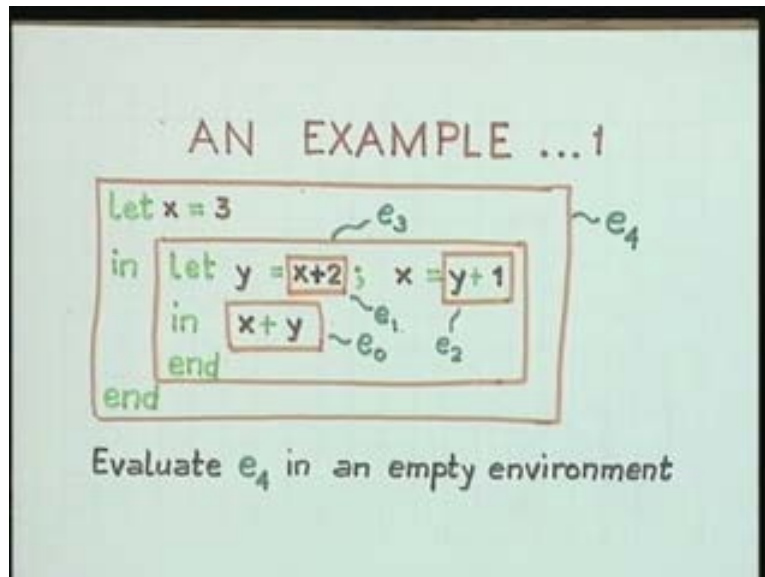


```
let x = 3 ; x = x + 1
in  x * x
end
```

The program is not hard to analyze but what you require to do are a few examples to make sure that your rules are right and that you do not get any configuration that is stuck.

Rules also give you a reasoning mechanism which you can do in a top down fashion or a bottom up fashion. Let us do it top down.

(Refer Slide Time: 40:20)

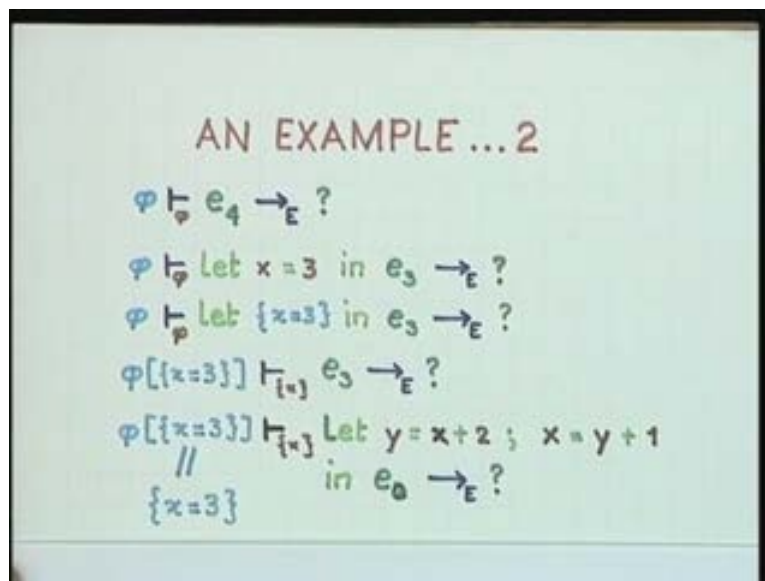


Essentially, you want to know what  $e_4$  goes to in the empty environment  $\phi$  with no predefined set of variables. Assume there is an empty set of variables  $\phi$ . The ' $e_4$ ' is an expression and it has a transition in the expression transition system. You also want to know what it does. You would like to know where a given transition goes to. If you have the answer to any of the transitions you have the answer to the final transition. You can think of it either as a top down recursively questioning or if you know the final answer you can think of it as a theorem that you have to prove with an intermediate step.

Supposing you know the value in the first transition then it is like a theorem that you can prove. I can prove that the first transition goes to a value 11 provided I can prove that the next transition goes to the value 11 by an application of some of the rules specified. Similarly, I can prove that the next transition goes to the value 11 provided I can prove that the transition after it goes to the value 11 etc. So, the transition depends upon evaluations in the updated environment and finding out what  $e_3$  goes to. The  $\phi$  updated with  $x = 3$  is just the environment  $x = 3$ .

You want to know in the environment  $x=3$  with  $x$  as a single variable in the environment what the next expression yields and this process is tedious because it is just pure symbol manipulation. If you are strictly applying the rules that we have given, it is a very tedious process. But the whole point is that it hides an execution of an algorithm in it somewhere and it is really meant to be done by a machine.

(Refer Slide Time: 43:26)

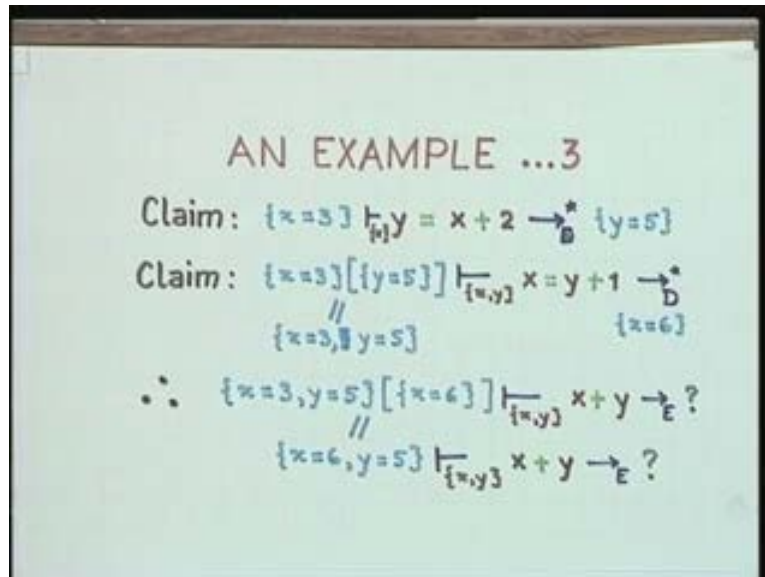


Take two claims which are really obvious and can be proved using the rules that in the environment,  $x = 3$   $y = x + 2$  yields environment  $\{y = 5\}$ . The claim might be proved in 0 or more steps of proof of the original transition systems for expressions and declarations. You may or may not be able to prove it in a single step of proof but in some finite number of steps you can prove it by a purely mechanical process of application of the rules. There is a further fact which I will highlight later. The next claim is that if you have  $x=3$  updated with  $y=5$  which is of course equal to the environment consisting of  $x = 3$  and  $y = 5$  with the variables  $x$  and  $y$  then  $x = y + 1$  yields the environment  $x = 6$  and it can again be proved in a purely mechanical fashion.

Finally we might conclude that the main body of the expression really has to be evaluated in the environment  $x = 6$   $y = 5$  which is of course obtained by the fact that  $x$  has been

redefined. So, you are considering an environment  $x = 3 \ y = 5$  updated with  $x = 6$  and the result of that updation is just  $\{x = 6, y = 5\}$  and it yields 11. The whole problem boils down to just evaluating  $x + y$  in the environment and that finally yields 11.

(Refer Slide Time: 46:09)



In all cases assume that the environment is duplicated in all the steps before the tag.

If you just work back 11 and fill up all those question marks that were given earlier with 11 then you have a top down proof. What is a top down proof? I have a theorem statement and I prove that statement by saying that a theorem is proved if I can prove the next statement down to the last line.

You can prove a theorem also as a sequence of goals to be achieved till you have finally achieved the goal or you can first make a guess about what you claim is to be achieved and actually prove it in a normal manner but we have used these alternate forms of proofs in mathematics always. In many trigonometric proofs you start with 'we have to prove a statement and therefore we can prove it provided we can prove the next statement...etc'. You start with a left hand side and you start with a right hand side and make them meet, but it is not a good presentation of proof. However, we know that that proof can always be converted into a rigorous proof once you know the final answer.

(Refer Slide Time: 47:19)

AN EXAMPLE ... 4

$$\begin{array}{l} \{x=6, y=5\} \vdash_{\{x,y\}} x + y \\ \rightarrow_E \quad \vdash_{\{x,y\}} 6 + y \\ \rightarrow_E \quad \vdash_{\{x,y\}} 6 + 5 \\ \rightarrow_E \quad \vdash_{\{x,y\}} 11 \\ \cdot \quad \boxed{\therefore \varphi \vdash_{\varphi} e_4 \rightarrow_E 11} \end{array}$$

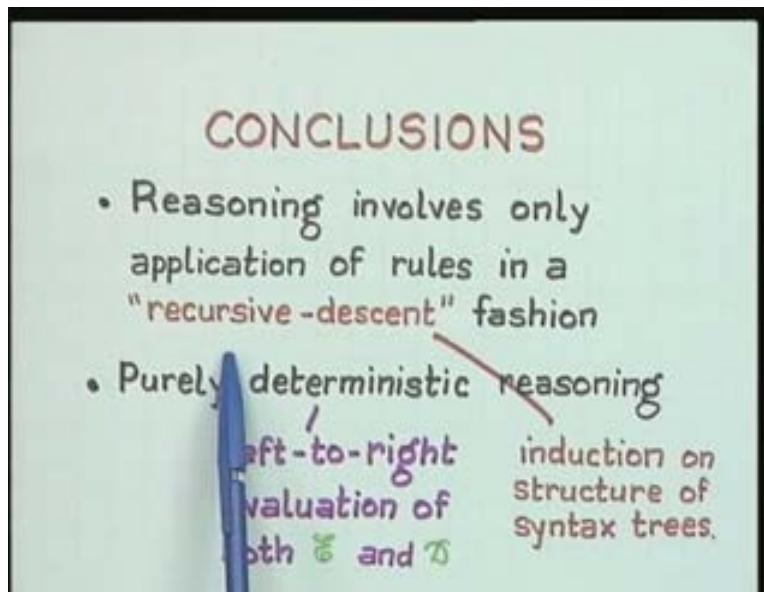
It is really in that spirit that you can look upon the rules as giving you rules for proof. More importantly, given a certain set of assumptions there are probably lots of theorems that you can prove. The important point here is that these rules are purely deterministic. In fact if you look at any individual step you know exactly which rule you have to apply whether you are going about it in a top down fashion of proof or a bottom up fashion of proof. It is purely deterministic and in that sense it is very different from a mathematical proof.

If it is deterministic it means that, if you can somehow encode the rules as algorithmic productions, then you have a deterministic algorithm which will give you unique answers. The form of these rules is like a mathematical theory. The forms of proofs that you want to do manually are also like that but at any point in that proof there is only one rule available and it is deterministic. It is predetermined what rule you should apply in order to obtain the next goal from the previous goal. In that sense it is not just a proof to be done by human beings, it has at least the potential for being implemented in a very deterministic manner and what actually enables this determinism here is the fact that our transition systems are deterministic.



The rules allow only left to right evaluation of expressions, only left to right evaluation of sequential declarations and that itself makes them deterministic and that makes the proof process also deterministic. Given a conclusion there is actually a unique proof which is not necessarily true of a mathematical theory. This uniqueness is guaranteed by these two points. One is the deterministic nature of our rules and the second is that you are always doing only induction on the syntax or in the case of the rules we are doing induction on the extended syntax where we included row also in the syntax. You are doing only induction on the syntax and there is only one way of obtaining a proof if your rules are deterministic and that means that you can actually evaluate it by an algorithm.

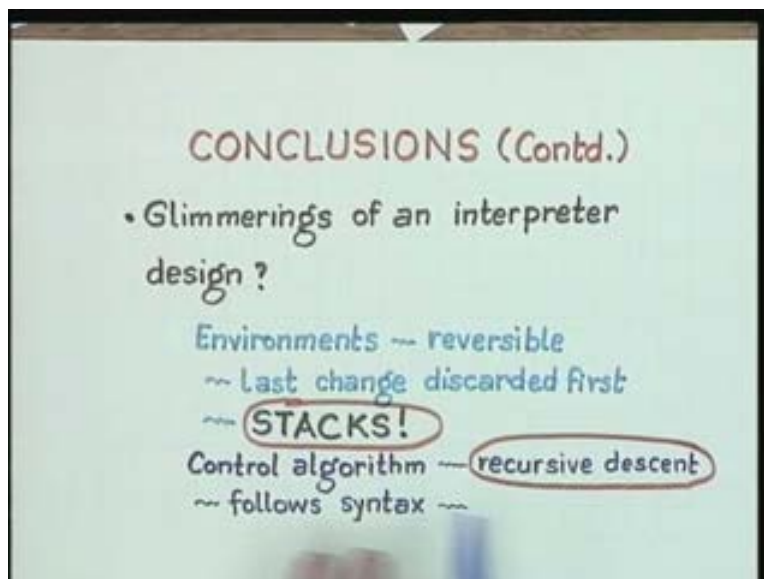
(Refer Slide Time: 51:02)



What makes theorem proving a difficult task is that given a set of assumptions you do not know what theorem your mechanical theorem prover can prove. So, the theorem prover has to be guided at all points of non determinism in order to give you a deterministic answer to work towards the goal that you are interested in. Most theorem provers are interactive because there are two kinds of non determinism. If you did not give it a goal it might prove some theorem but that may not be what you want. If you gave it a goal there might be too many different ways of proving it and so it requires a user to guide it along some path.

In the case we have taken up the rules are such that it is purely deterministic; there is only a single way of reasoning by induction on the syntax trees and therefore it has a potential of having a deterministic algorithm. It means that the rules themselves give you essentially an interpreter design where you want reversible changes in environments. Data structure, which allows for reversible changes in environments, is a stack but that is a matter of implementation. That is not preached by the semantics and the fact that it is all done by induction on the syntax tree means that you can implement the control algorithm, either the interpretation or the code generation followed by interpretation both in a recursive descent manner as part of your parser for the language. Except for the fact that the parser of the language has a more refined grammar which takes into account various factors that you will have to patch up, you have the glimmerings of an algorithm to implement.

(Refer Slide Time: 54:15)



Note that our semantics is defined on the syntax trees and not on the actual syntax that the parser uses, which is a much more refined and sophisticated grammar than what we are considering in our semantics. It means that the main glitch in going to the algorithm is to determine exactly in which point of that parsing routine you will have to include the code generation. You might have to distribute the code generation across various routines in your recursive descent parser.