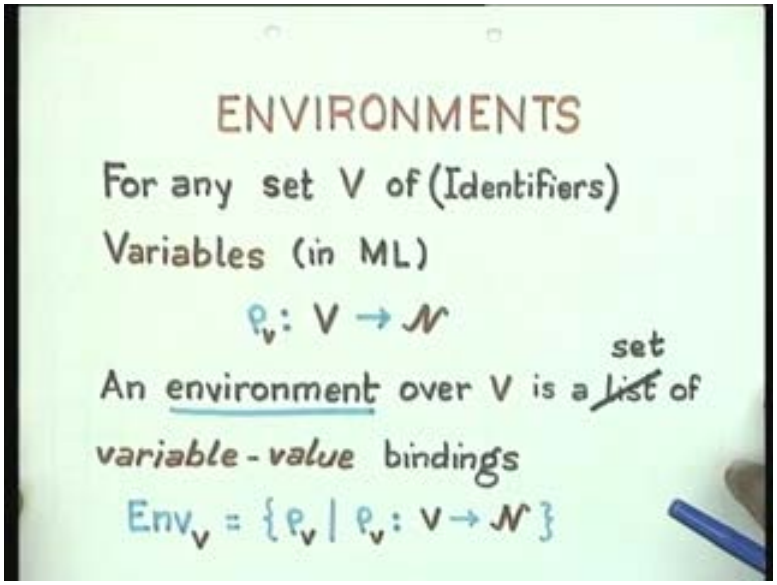
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 15
Lecture Title: Summary

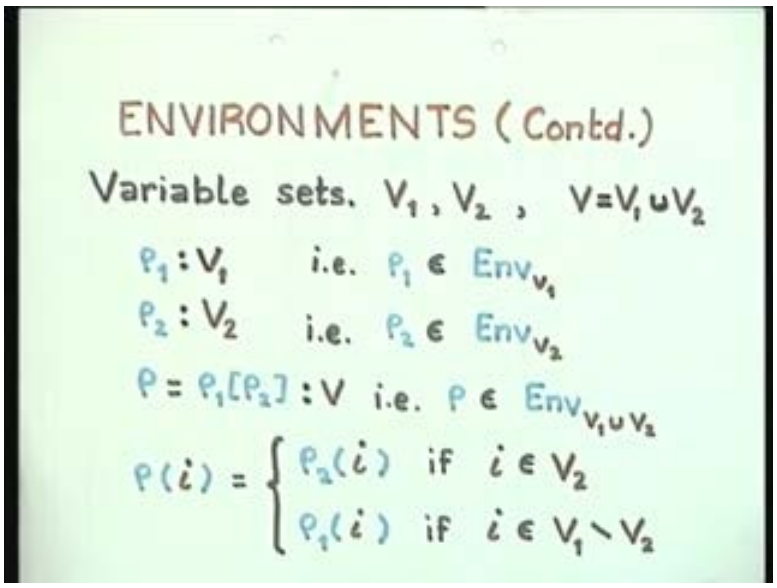
Welcome to lecture 15. I will just summarize some of the concepts that we have done so far and prepare ground for the future.

(Refer Slide Time: 00:40)



We started with the notion of environments. We decided that environments are some variable-value bindings for an ML like expression language with declarations. Then we also looked at some possibilities for reversible changes in environments.

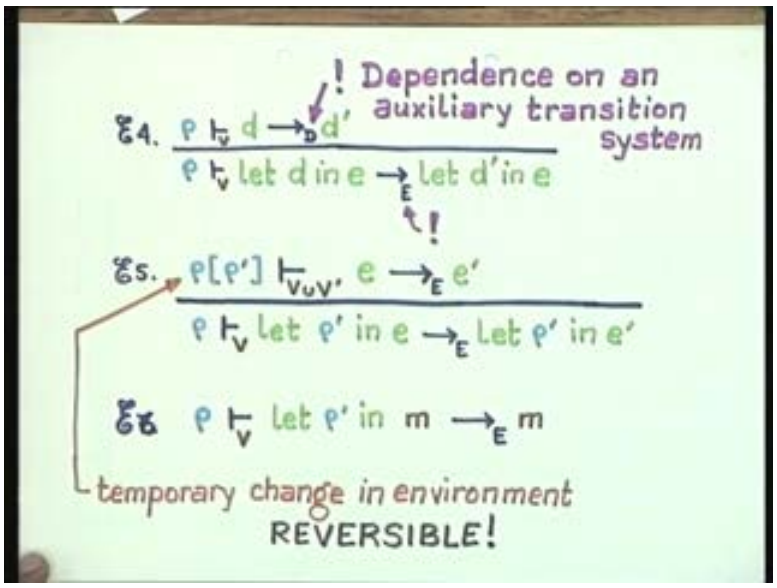
(Refer Slide Time: 00:55)



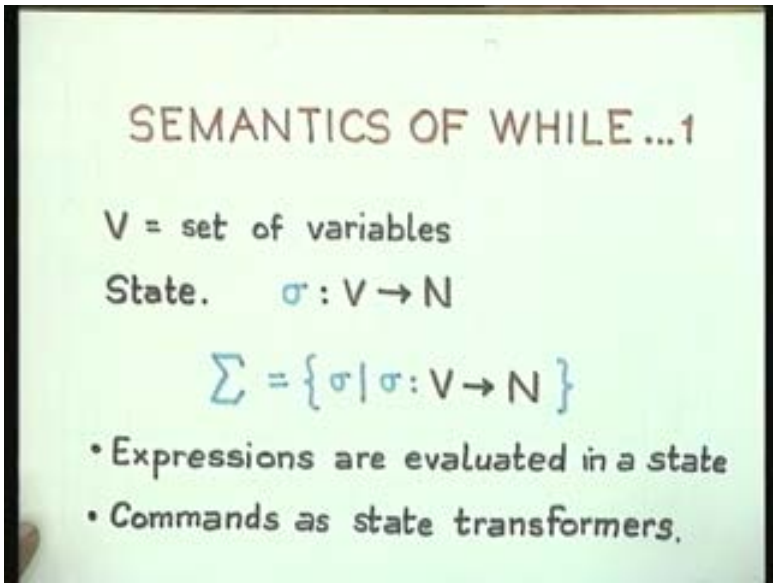
We discussed environment updation and we also looked at how these environment updations are always of a reversible nature for example, in the expression language with the ‘let’ construct as in ML. We were evaluating an expression in some original environment row and if there is a declaration inside it what it means is that we temporarily update the environment to a new little environment.

The processing of the declaration gives you a new little environment in which you evaluate this expression but eventually you revert to the original environment and these changes are reversible. On the other hand while looking at a simple imperative language we first took a simple view. We defined an abstract concept of a state and we defined that also as these variable-value bindings and we defined the semantics using this state transformation as an updation mechanism. We were of course interested in modeling state changes.

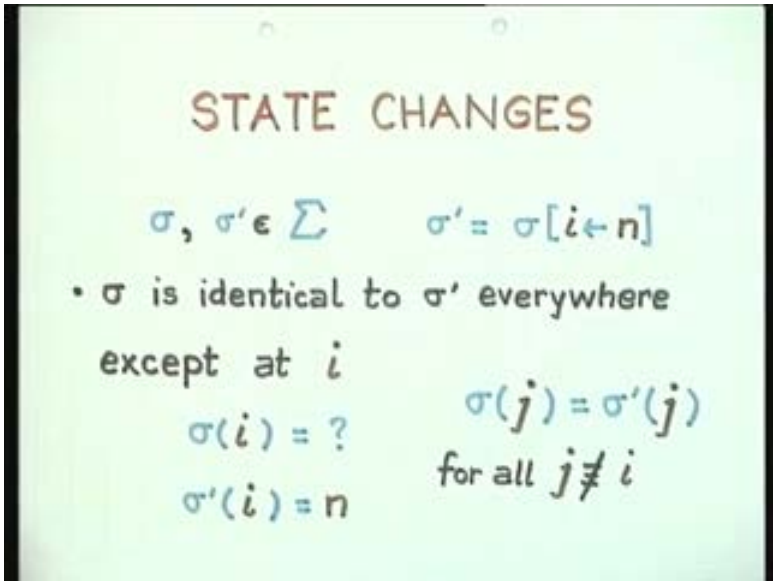
(Refer Slide Time: 01:41)



(Refer Slide Time: 02:07)

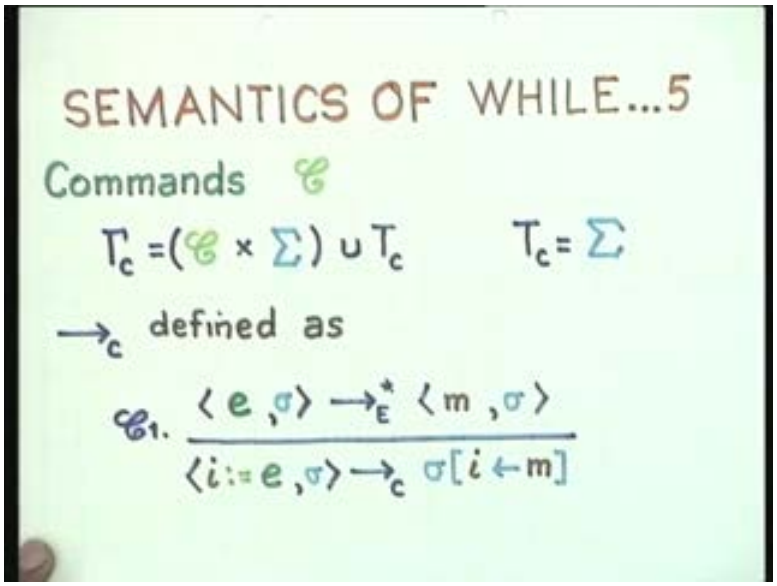


(Refer Slide Time: 02:25)



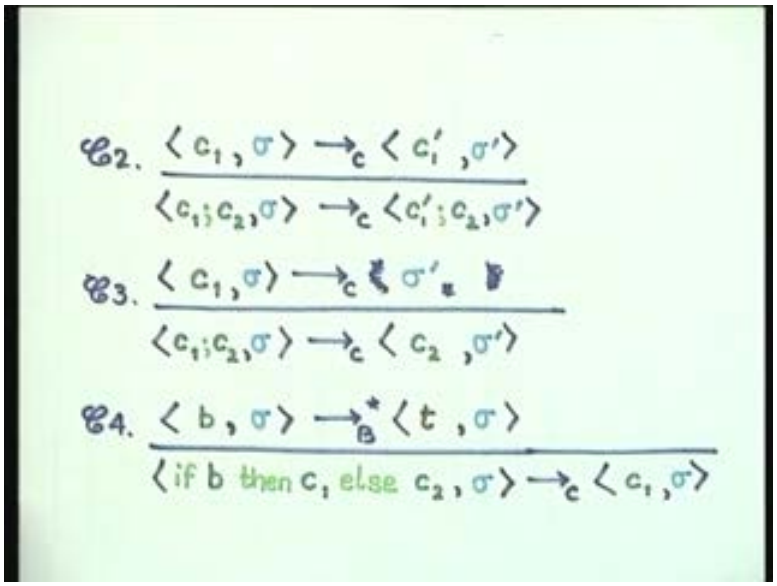
We have a state updation notion, which looks a lot like the environment updation notion, but is actually quite different because it leads to irreversible changes. We also defined the semantics of commands with an underlying expression language in which this for example is a permanent state change. Wherever you have an assignment statement there is a possibility of an irreversible state change.

(Refer Slide Time: 02:58)



We discussed the various control structures and the most elementary control structures namely sequential composition. We gave an inductive definition for that and always when we are talking about commands we are talking about the possibility of state change and other control structures like the conditional control structure and the looping control structure.

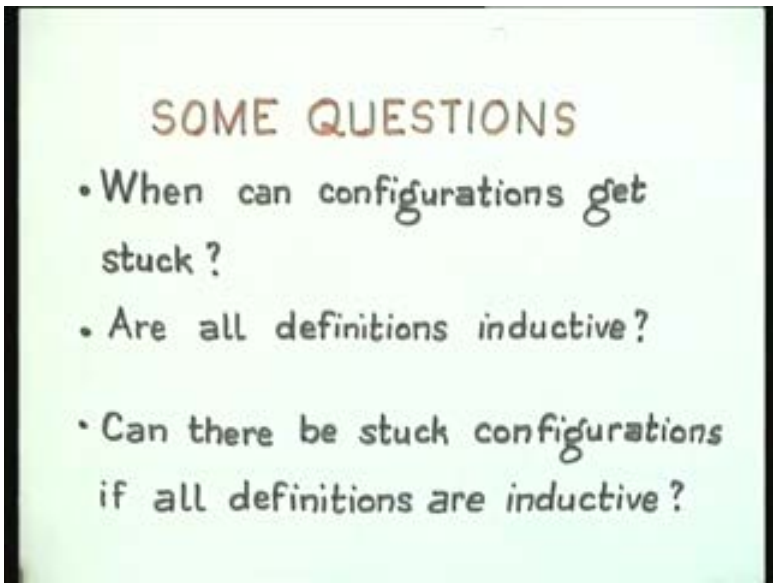
(Refer Slide Time: 03:21)



In our general discussion on transition systems we also have to keep certain points in mind which is independent of programming languages. These are some general questions like when can configurations get stuck and you might need to know that even while modeling other systems which are not necessarily programming-language based.

Then we looked at the case of formal languages. By formal languages I mean programming languages and specification languages or any language that has a formal grammar which can be completely defined by our grammar. Are these definitions inductive? If your definitions are all inductive then you have a certain level of confidence that most of your configurations will not be stuck because then you can use the induction process to prove that some of your non terminal configurations are not stuck.

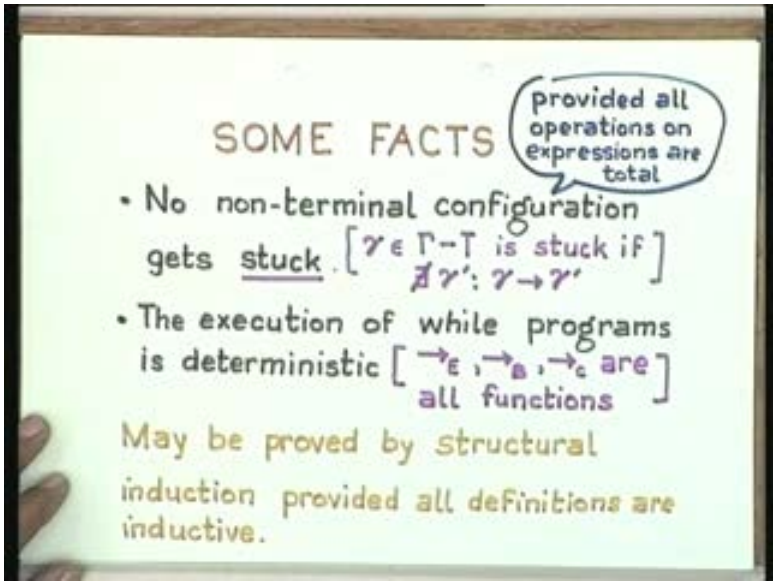
(Refer Slide Time: 04:55)



But there could be stuck configurations even if all your definitions are inductive and in programming languages what it usually means is that this would lead to some form of runtime error and not compile time. In our definitions for example, we have not explicitly considered various binary operations. We have just considered a general binary operation but if you were to take division as a binary operation then the corresponding axiom for division would have to include a side condition that you are never dividing by 0 and that means that there are configurations in which an expression might contain a division by 0 which would be stuck for which you would not able to apply these axioms or rules of influence.

Provided all our expressions are total functions (meaning they are not undefined for any particular value of the arguments) then you can show that no non-terminal configuration ever gets stuck and you can show that the execution of while programs is deterministic. So far we are looking only at deterministic programming languages. In fact we will never actually look at nondeterministic programming languages but it is important for us to know that deterministic in this case means that our transition relation is really a function which it need not have been to start with.

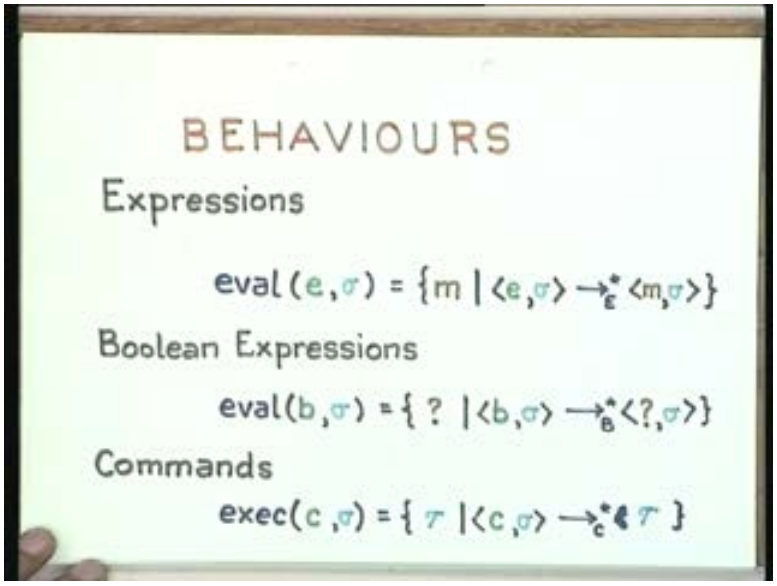
(Refer Slide Time: 06:34)



A powerful proof technique for proving properties about programs is an induction method. Also one obligation is that you have to also define what is meant by a behavior and we defined behavior of expressions in terms of an evaluation. In the imperative language if in a given state the expression can evaluate to some terminal configuration then the value in that terminal configuration is taken to be the result of evaluating this expression in that state.

In general as I said since we are not assured that the transition relation is a function this means that you could have more than one possible result for the same expression in the same state. However, once you have proved that your transition system for the language is deterministic the result of this would be a singleton set and of course sometimes non empty and sometimes empty but that is when you consider more complicated cases where the expression could be recursive and therefore there could be an infinite recursion or the expression could have commands embedded in for which there are infinite ‘while’ loops etc.

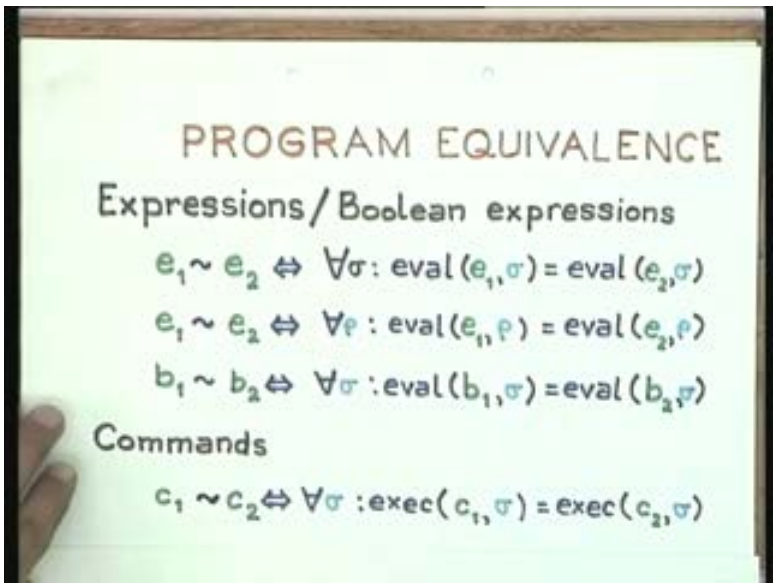
(Refer Slide Time: 08:45)



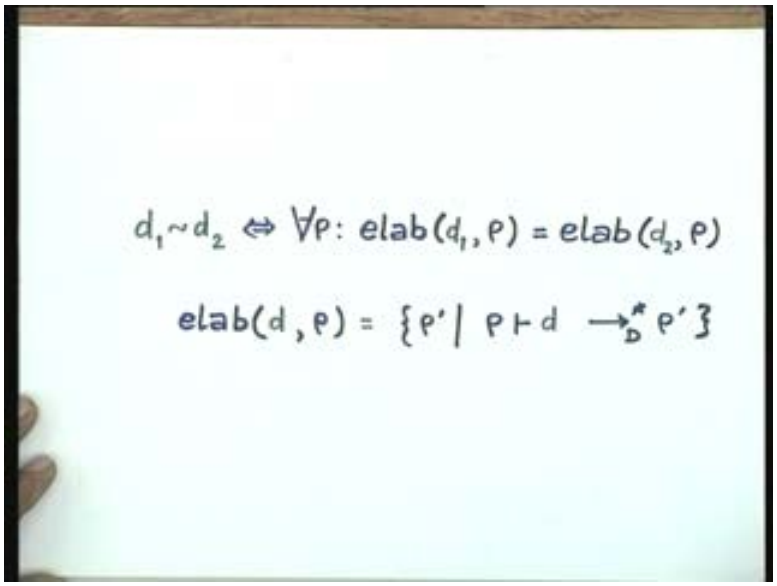
Then similarly for Boolean expressions we defined the notion of evaluation and for commands we defined the notion of an execution and commands yield states and since our transition relation is deterministic every command defines a function from state to state (from an input state to an output state) and what it also means is that it also confirms the view that commands are state transformers.

Having defined the notion of a behavior we defined program equivalence. Two expressions are equivalent if and only if under all states or under all environments they yield the same results. This of course has to be taken with a pinch of salt. Two commands are equivalent or two programs are equivalent if and only if under all input states their executions yield identical final states.

(Refer Slide Time: 09:25)



(Refer Slide Time: 09:40)

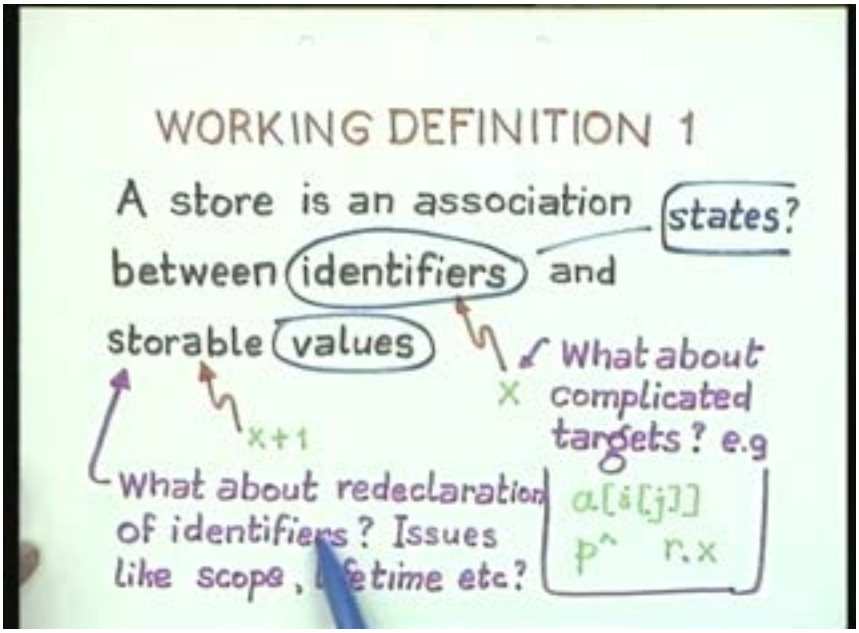


We also looked at the futility of actually trying to give a definition for equivalences of declarations. Then our main problem was that we had defined the state as an association between identifiers and values in our while programming language and then that brings up various other questions. When you actually want to extend the language to other constructs then it turns out that for example; the assignment is the most important form of an imperative command, a state changing command and all other commands which

change state in visible or invisible fashions can be modeled in terms of sequences of assignment statements. This is a common practice.

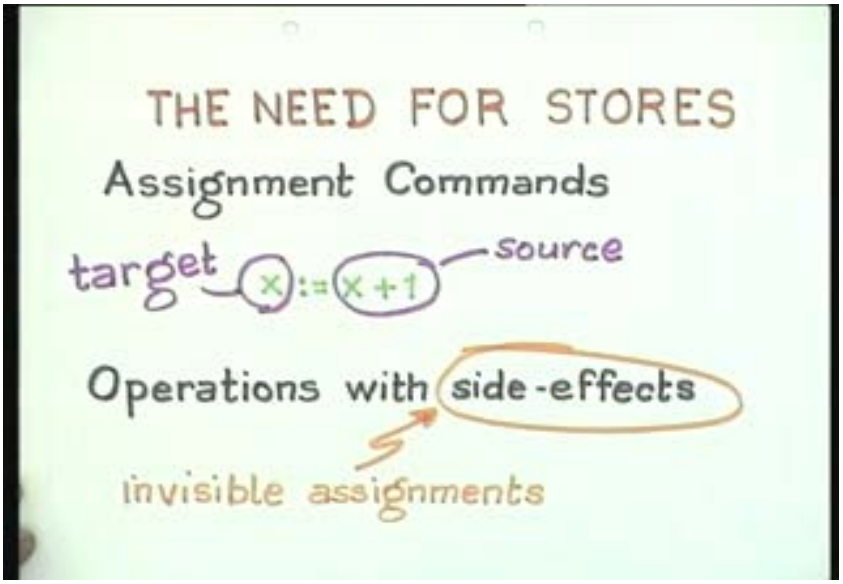
So, the assignment statement is very important and it turns out that the notion of state is rather weak because firstly the left hand side of an assignment is not necessarily always just an identifier for it could be an expression. Secondly, identifiers are re-declared and there is a reversible process also when you consider scope issues, lifetimes of identifiers etc.

(Refer Slide Time: 11:23)



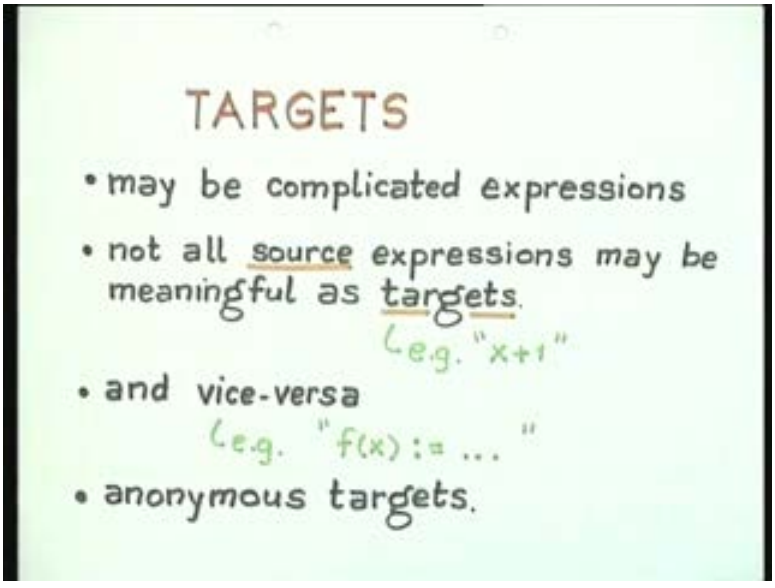
Re-declared identifiers means that they are actually two different identifiers carrying the same name and then there are issues like scope, lifetime and extent which is why with a simplistic notion like a state it is not really possible to capture that. Also the notion of side effects and what happens in side effects is also quite difficult to capture in the case of states because of the fact that many different objects might actually have the same name but the same object could also have many different names due to aliasing and therefore in the case of aliasing when the same object has many different names then what you have as side effects are essentially invisible assignments.

(Refer Slide Time: 12:13)



You make an assignment to one name of that object but that means that you have also simultaneously made assignments to all other names of that object too. There are all kinds of invisible effects which complicate matters.

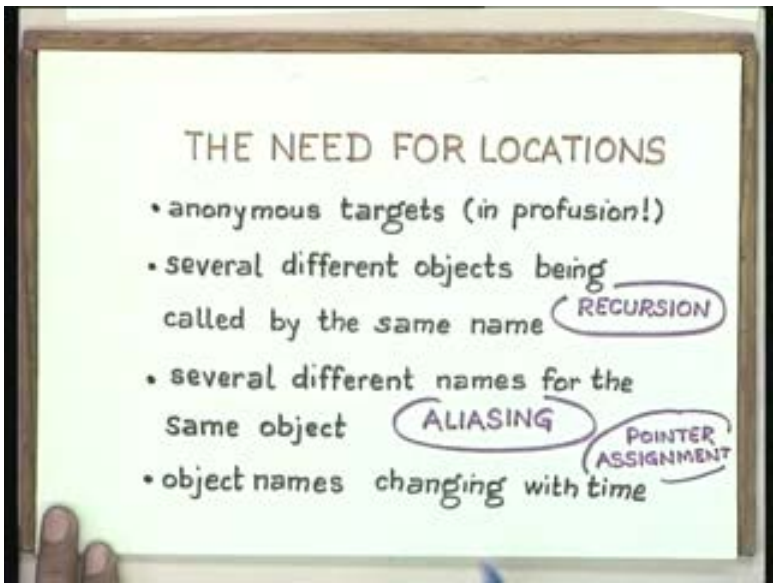
(Refer Slide Time: 12:42)



In general when you look at the assignment statement you might have complicated expressions and also there is a restricted language meaning not all source expressions are meaningful; not all target expressions are meaningful, the set of meaningful source expressions is not disjoint from the set of target expressions, there are anonymous target expressions, anonymous objects which are created and names that keep getting transferred as new objects are created and old objects lose their names.

What it all boils down to is that when you have anything like an assignment statement in a programming language then the implicit part of that assignment is that there is some need for a new entity called locations which somehow is a model of memory in most digital computers. If you look at all these problems it is necessary to identify this notion of this object and we have object creation in several different ways temporary or permanent and we have updation of its values. We require something in between an identifier and a value and for that we require the existence of a new domain of objects called locations which are always new.

(Refer Slide Time: 14:20)



Whenever a new object is created it has a new location and the new location has a content and particularly when you have anonymous objects and especially names changing from

one object to another during the process of the creation of new objects, we require to locate the old object too even though it has lost its name. In this process it means that our notion of a state which was very close to the notion of an environment really has to undergo some drastic revisions. Firstly, we require to model irreversible changes and we have to distinguish between environments and a set of objects which are distinguished by this process. So, we will look at locations as somewhere intermediate between identifiers and values and then our declarations in our imperative language automatically take a certain meaning. So, constant declarations would just be treated like the variables of ML.

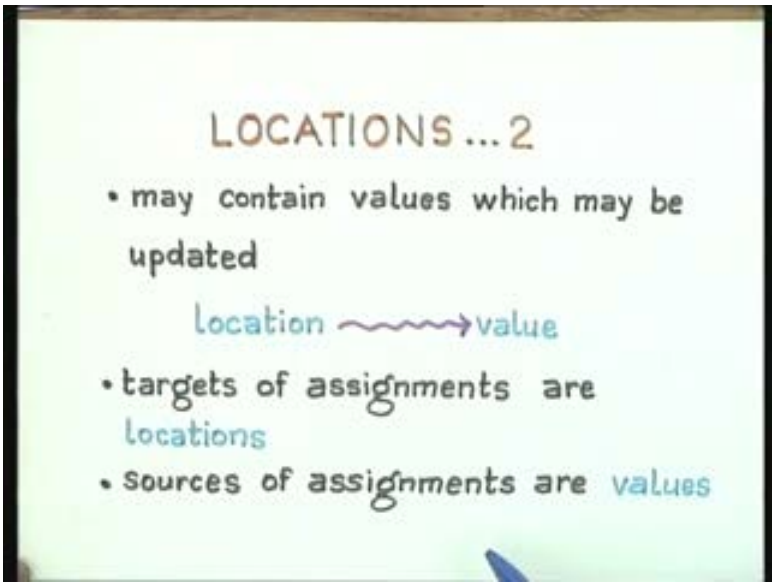
(Refer Slide Time: 15:30)

They would just be identifier-value bindings whereas variable declarations would actually be identifier-location bindings. During program execution you could have new names; you could have the same name transferred to several objects but all those objects would have different locations. At least you could look at locations somehow as identifying the basic structure of an object during execution.

(Refer Slide Time: 16:40)

We will define variables as identifier-location bindings and locations as containing values. So, there is also a location-value association which is not really a binding. The word binding is normally used for declarations. We will look upon the targets of assignments only as locations and the sources of assignments as the values contained in locations.

(Refer Slide Time: 17:30)



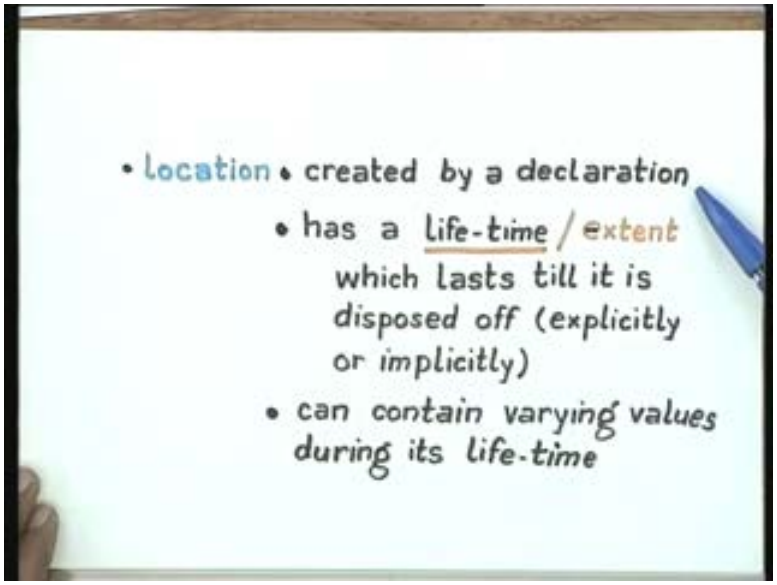
You can think of a location as containing a value which can be changed during command execution but during the lifetime of this location there is an identifier to location binding which comes from a declaration. You can have fairly ephemeral creations like that. Let us assume that new locations are always created by declarations. Every location has a lifetime or an extent which usually lasts till that location is explicitly or implicitly disposed off. The fact that a location is created by a declaration means that it does not necessarily last forever because a declaration creates a little environment that has a lifetime which is usually governed by the scope of that declaration.

What constitutes scope in a program text works out at runtime to a lifetime for each object that is created in that scope. So, each object that a declaration has is created when that declaration is elaborated and it has a lifetime which lasts till the end of that scope. At the end of the scope it is automatically disposed off. Of course it is possible for example in the case of pointer based creations to dynamically create new objects and new locations that are created with typical statements like new P for example and which get disposed off also explicitly. Locations might be created either through declarations or through commands and they might be disposed off and they have a finite lifetime which might depend upon whether they are explicitly or implicitly disposed off. When they are implicitly disposed off we are usually talking about a reversible change and when they are explicitly disposed off we are talking about an irreversible change.

Instead of assuming that we have an unlimited number of identifiers which can always be used because of the problems associated with having the same identifier for several different objects and the same object having several different identifiers, we will assume that what we have as unbounded is really a number of locations. Every time a location is created there is no conflict between that location and anything else that has been created before, which still is living and not dead. In fact we will go further and say that every time we create a location, we create a brand new location which has nothing to do with any location that was created in the past. We really cannot say the same for identifiers.

Locations actually take the roll of an unbounded number of identifiers that we had in the state. A refinement of the concept of state is that you assume at any stage that you have only a bounded number of identifiers but because when you elaborate declarations you get only a bounded number of identifiers and you require only a bounded number of identifiers. However, because of the fact that you might be dynamically creating new objects to which you might bind the existing bounded number of identifiers what you require really are an unbounded number of locations. In practical terms it means that you are really assuming that you have got an unbounded amount of memory in modeling the semantics of the language, which is not completely realistic. But it is an ideal which can be refined later when you define an implementation document with limits imposed on them.

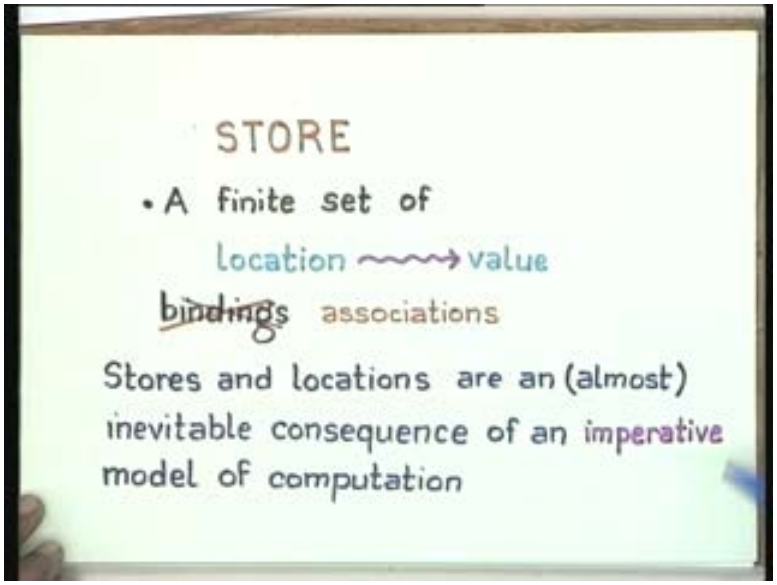
(Refer Slide Time: 22:34)



When you actually have a finite memory and you have to model the language then it is a matter of doing a certain amount of exception handling. Otherwise in an ideal environment you assume that you have a computational environment in which you have an unbounded number of resources but at any stage in the computation you have only a finite number of identifiers to deal with. These locations can contain of course varying values during their lifetime.

We will define a store as just a finite set of location to value associations. We will not use the word bindings there because we are talking about associations which can change. This concept is not very surprising or it is almost an inevitable consequence of having an imperative model of computation. So, it really depends upon the model of computation that you are really trying to deal with and any kind of imperative model of computation in which there are objects that can be updated irreversibly, brings in almost the concept of locations.

(Refer Slide Time: 23:35)

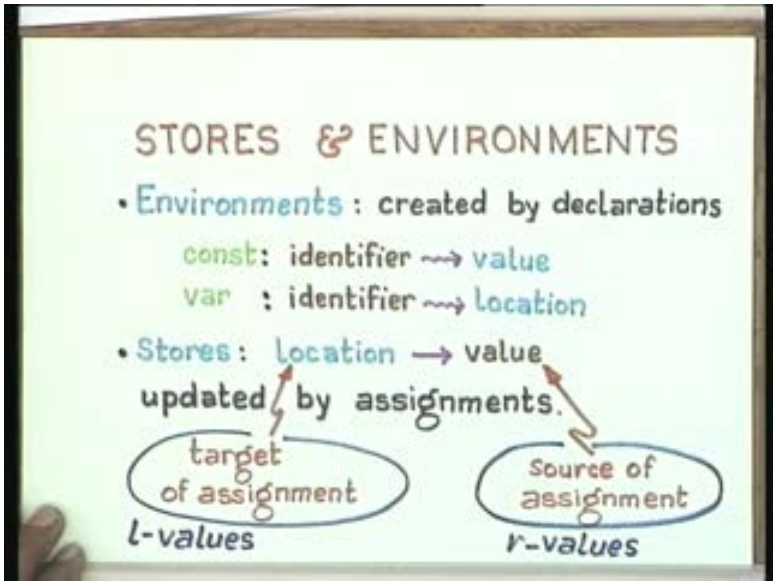


We will look upon environments as being created by declarations and in particular we will look upon constant declarations as identifier-value associations with no locations in between. We will look upon variables as identifier-location bindings and stores which are a refinement of the state concept would just be location-value assignments.

The important problem with the assignment is that the left hand side of an assignment is really a target. The target of an assignment as I said is a restricted class of expressions and the sources of assignments are also restricted classes of assignments. But the important distinction we will make is that the sources are all values and the targets are all

locations. These are called L values and R values and once you bring in the notion of stores there is the concept of what kinds of values you can put in locations.

(Refer Slide Time: 25:31)



We will look upon storable values for the present. Firstly, you could have constants in them and then you could also have locations themselves as storable values and lastly because of the Von Neumann sort of architecture that is an implicit underlying architecture of all the imperative languages, your storable values could also be programs. In fact this is the only way you can account for self modifying programs.

Typical examples are the s expressions of LISP. It usually means that in practical terms it is actually very simple. You have a piece of code which actually accesses itself and modifies it and then executes it. But logically you are looking at it as a problem of a function or a program being applied to itself and a function application to itself is logically unsound.

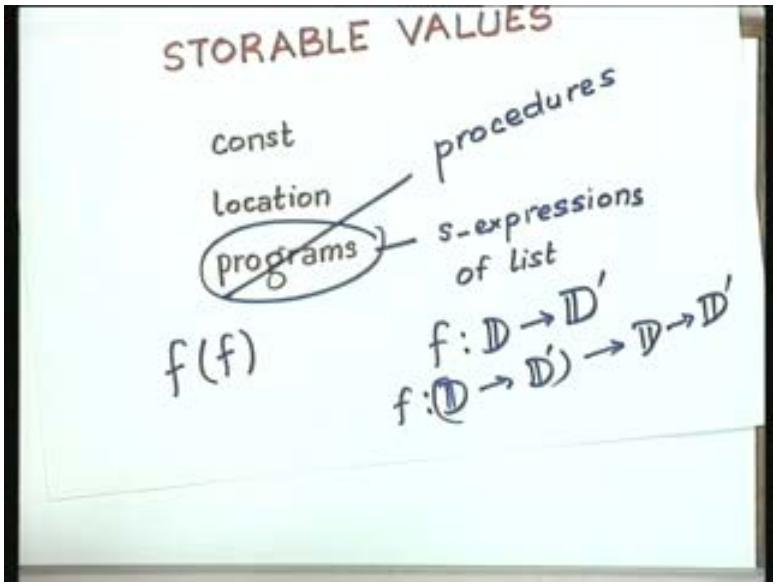
However, with the notion of locations and storable values all these can be given a meaningful interpretation without actually getting into any contradictions.

It is mathematically unsound to think of a function f being applied to itself, $f(f)$ but if you look at some of these self modifying programs written in LISP or even assembly language you can load the entire assembly program starting from some location and you can actually access that location and modify elements and then transfer control to execute the modified code. While it seems practically quite feasible, if your transition system has a transition relation which is actually a function, then all this is tantamount to a function being applied to itself. A function has a certain type but how can you actually apply it to either itself or to a modified form. That means if your function was originally defined as $f : D -> D'$ for such an application this function $f : D -> D'$ also has a type that can go from

$f : D -> D' -> D -> D'$, which seems contradictory.

So, a mathematician completely avoids these contradictions but somehow they seem to make programming sense because we know that it can be done and we know it is possible. It turns out that once you introduce locations you can actually look upon locations as being distinct from identifiers and you can assign types to locations and to identifiers in such a way that such functions become meaningful, if you provide a suitable mathematical structure.

(Refer Slide Time: 30:44)



It is enough for us to know that for any consistent theory which requires an explanation of all the kinds of different forms of object creation that we have seen, it is necessary to have locations and stores. When we do procedures we will also define procedures as being storable values just as programs could be and that is in fact the only way to explain transfers of control to procedures and to define the effect of procedures on certain values and the returning of certain values.