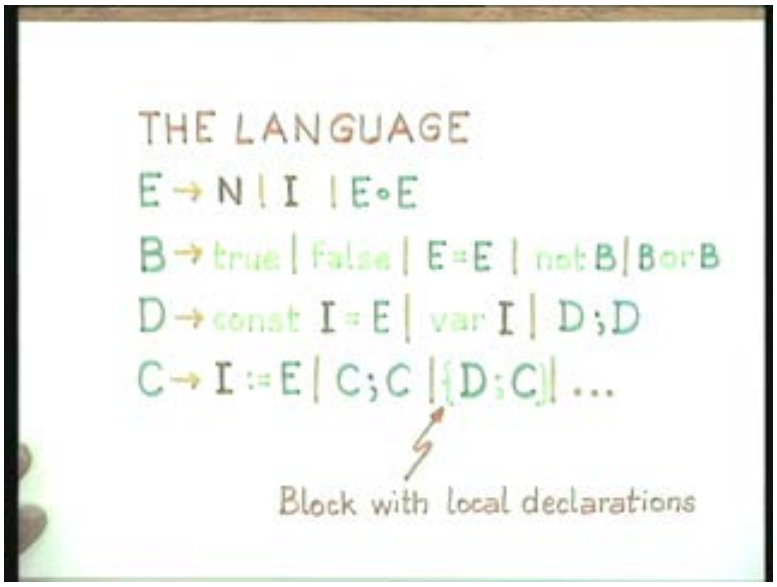
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

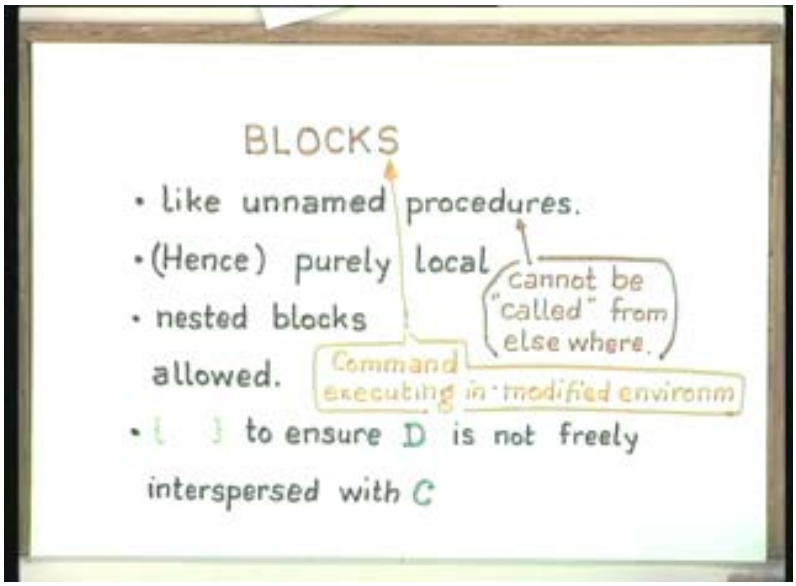
Lecture no 17
Lecture Title: Blocks

Welcome to lecture 17. We will discuss blocks but before that we will briefly recapitulate what we did in the last lecture. Here is the grammar of the language. The most important new construct is mixing declarations within commands and that creates a block with a local declaration and blocks are like unnamed procedures as far as this language is concerned. They are purely local and they cannot be called from elsewhere. However, you can nest blocks. We defined locations as an infinite collection.

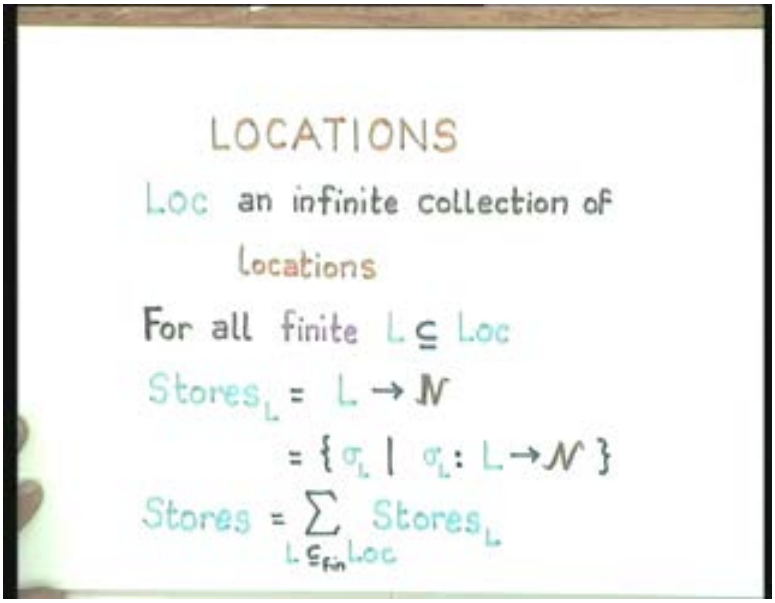
(Refer Slide Time: 00:35)



(Refer Slide Time: 00:55)

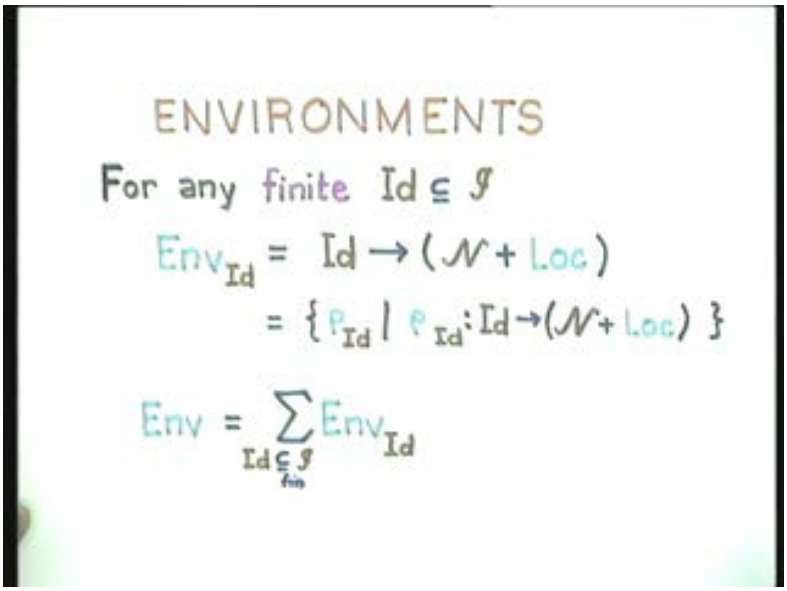


(Refer Slide Time: 01:10)

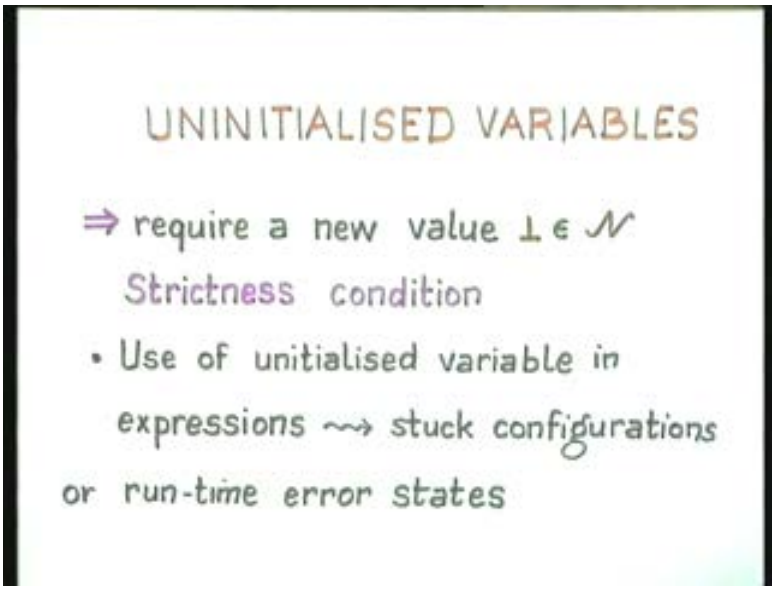


We defined ‘Loc’ as an infinite collection of locations and a store is a function from some finite subset of locations to values. A store over L is such a thing and the set of all stores is overall the finite subset of locations. Now locations become values that can be stored and so an environment is a mapping from identifiers to either values or locations where the identity as to whether it is a value or location is preserved in the mapping. I spoke briefly about uninitialized variables and the need to introduce a new undefined value which means that we also have to impose strictness conditions.

(Refer Slide Time: 01:50)

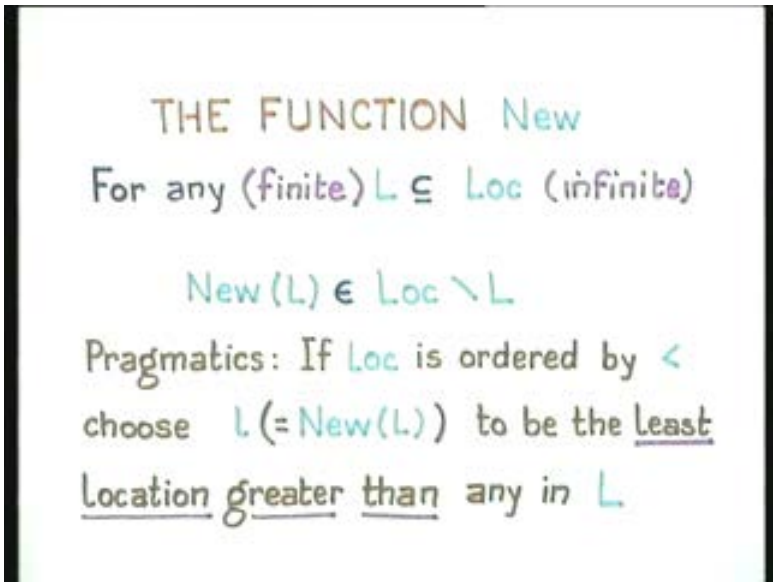


(Refer Slide Time: 02:10)



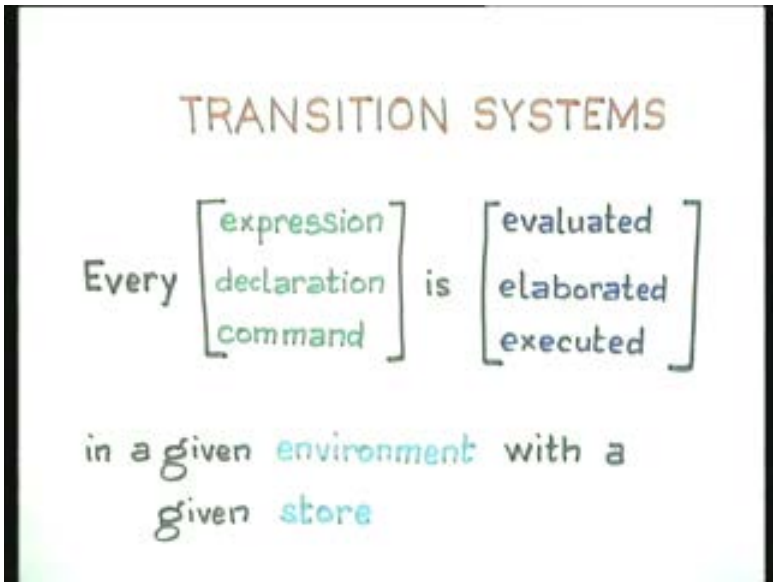
Expressions evaluated with some sub expression which leads to an undefined value would all be undefined and pragmatically speaking that would just lead to a runtime error. We said that we required a new function which given a finite collection of locations produces a new location from this infinite collection ‘Loc’ and there are very simple ways of implementing the availability of a new location.

(Refer Slide Time: 02:40)



Basic semantics in transition systems is that any expression, declaration or command is evaluated in a given environment with a given store. The state of a computation in such a language consists of both an environment and a store. The concept of state which has so far been sort of abstract is more concrete now. It consists of both an environment and a store.

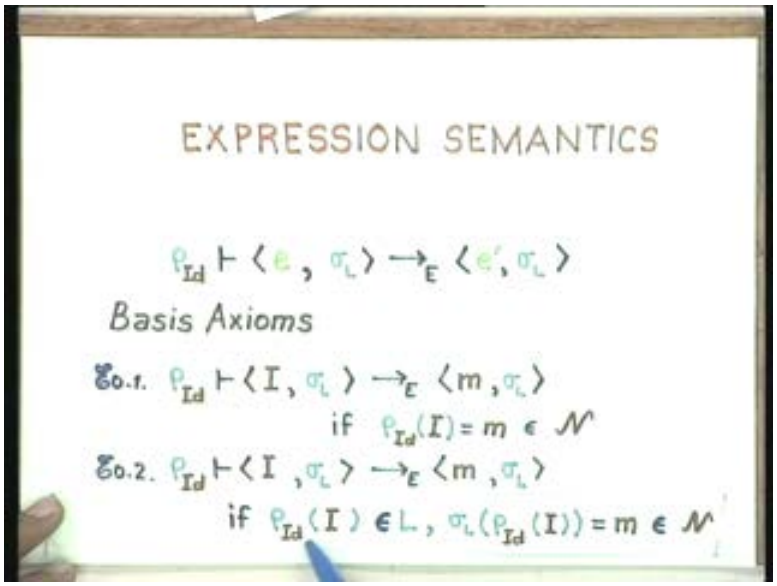
(Refer Slide Time: 03:00)



We are still talking about languages which are state based. Only the notion of state has to be more refined. In the expression semantics in this language we look upon all expressions as basically those occurring on the right hand side of assignment statements.

Here there are R value evaluations of expressions. Given any identifier if the environment points it out to be a constant value then that is what this evaluation gives us and if it is a variable then the environment should give you the location and the store for that location should give you the value.

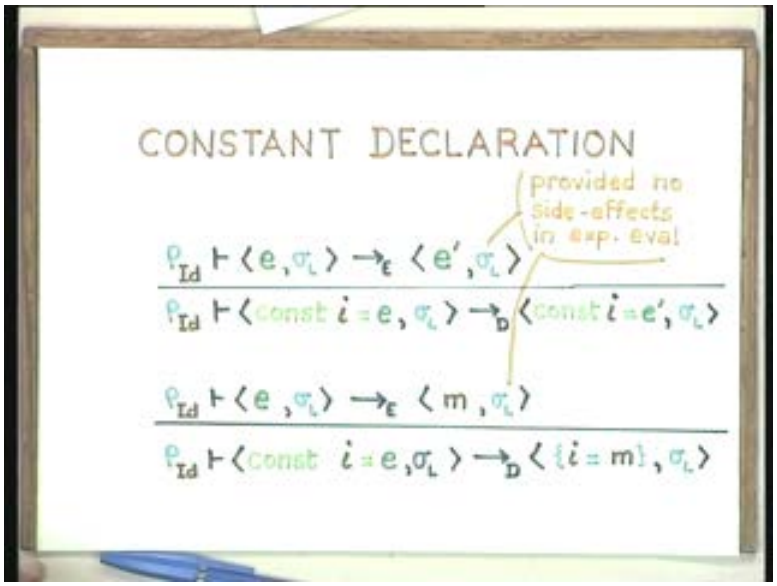
(Refer Slide Time: 04:15)



There is a level of indirection here for variables. Let us quickly look at constant declarations. If you assume that there are no side effects in expression evaluation in our language then you evaluate the expression through the expression language several times till it reaches a value and the effect of the constant declaration in a given store is to nearly create a new little environment with the same store.

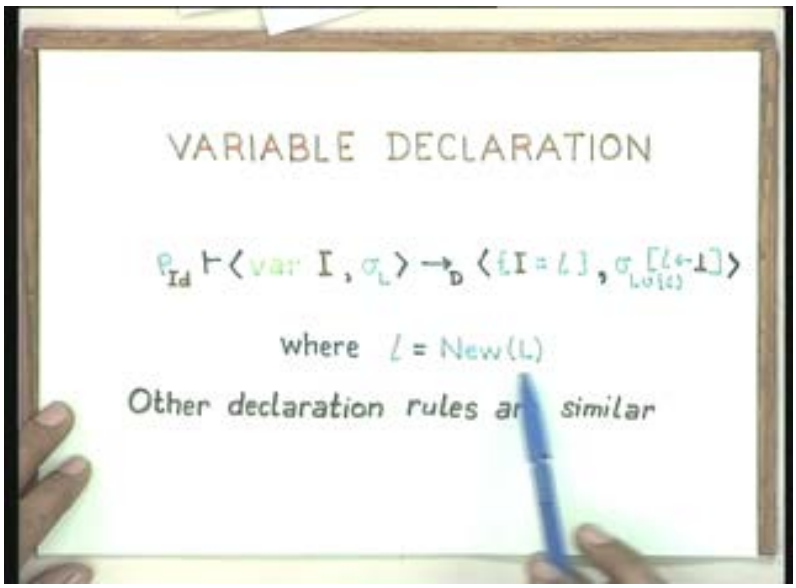
If expressions could have side effects then there is a very simple change that you have to make. It only means that you cannot assume that in the process of evaluation the store remains the same as when you started the evaluation. It means that with each application of a single step of the expression evaluation mechanism your initial store σ l goes to some σ l’ and it could keep on changing.

(Refer Slide Time: 06:07)



Correspondingly it means that the declaration also moves to σ L' and eventually when e σ L -> m σ L'. The declaration moves to the creation of this new little environment and a modified store σ L'. Under the assumption that there are no side effects in expression evaluation the stores are maintained otherwise in general they do not have to be maintained. A variable declaration on the other hand is anyway guaranteed to change the store because it really changes the very function by adding an extra location that from σ L to a σ L union. The new location and the effect of a variable declaration is to obtain a new location which hitherto is not present in L and bind the identifier to that location with a new initial value for this location which in the case of the particular language is an undefined value.

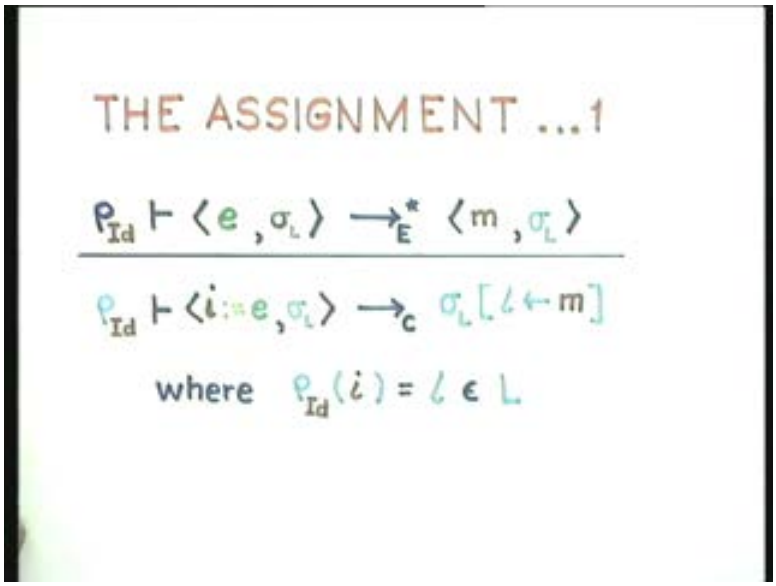
(Refer Slide Time: 06:50)



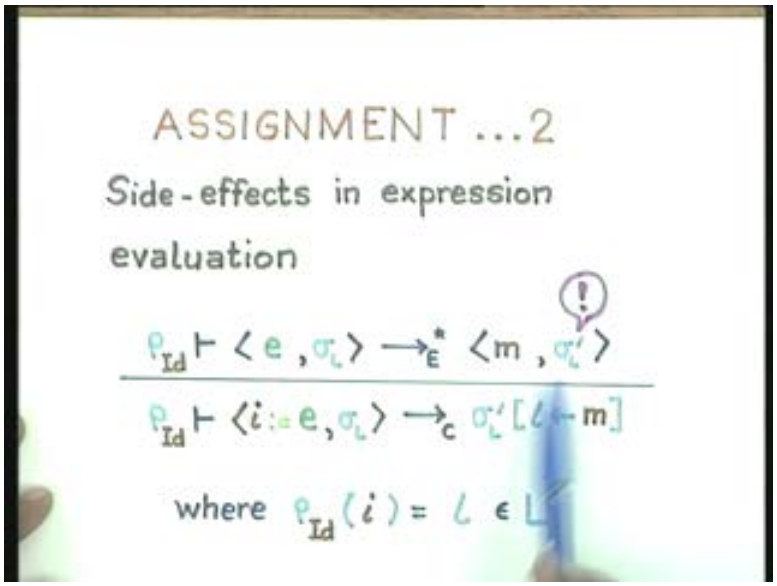
The other declaration rules are similar. The effect of an expression language is most importantly felt in the assignment. So, given an environment row and an assignment statement where I is assigned e again, we are assuming that there are no side effects which means that the evaluation of the expression in 0 or more steps (0 is important because this expression e could just be a constant value) could lead to some value m in the store σ L and the effect of this assignment is to just change the store to accommodate the new value m in L.

Whatever may have been the previous value contained in L is erased and a new value is stored where the environment gives you the location which has to be updated. If there are side effects in the expression evaluation in general, all that it means is that the store could also change in the process of expression evaluation. If you have a changed store from (σ L to σ L') in the process of expression evaluation then the new store is whatever is there in σ L' updated with this new value. The other expression rules, declaration rules and command rules are simple modifications to take care of environment and store from whatever we have previously done.

(Refer Slide Time: 08:06)



(Refer Slide Time: 08:27)



Let us look at blocks. We can look upon block semantics in this fashion. A block is a command in our language and the first step is to elaborate the declaration in the block. For the moment I am assuming that this declaration d is non empty. So, suppose there is the declaration d → d' and I am considering a more general setting, your initial store σ L could be changed in several ways. In the process of elaborating a declaration the store anyway gets changed because new locations might be added. Further if that declaration contains some expressions which could create side effects you could have changes in store because of that also.

The store could change for two reasons. One is that new locations are added. So, the L changes to L' and therefore the store changes but in addition if there are side effects in the expression evaluation mechanism then those changes could also be incorporated in this store and the result of this block declaration is to just move. The rule might be applied several times. Again as usual in our language for specifying the semantics we will introduce pseudo commands of this form, which are not there in the original language, just like we did in the case of declarations where we introduced environments also as part of that declaration syntax in order to facilitate an easy rule for evaluating declarations.

Similarly, for commands which have local declarations, we assume that an environment has been created and a new meta-syntactic category of this kind also exists in the command language because the purpose is only for specification. By the command rules the effect of the command C is executed in this temporarily updated environment.

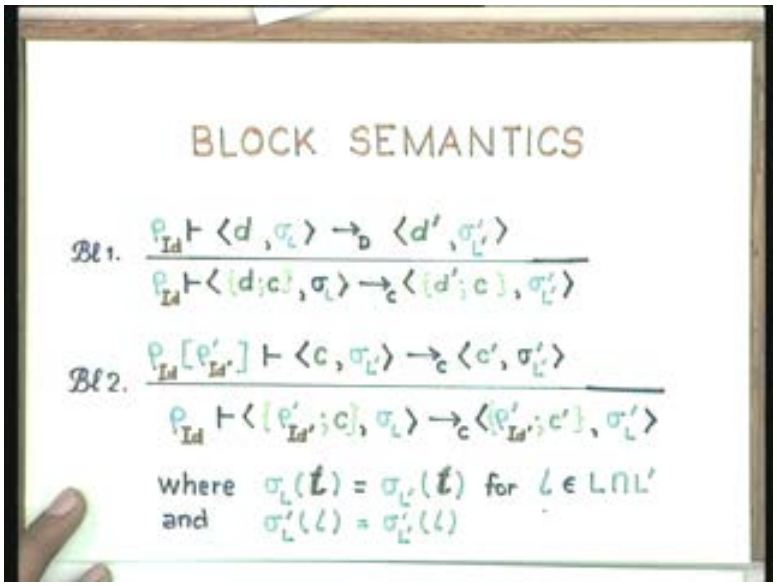
If the effect of a declaration after several moves in the declaration in semantics finally gives you a little environment row prime, then c is executed in this modified environment to reach some c' with some changes possibly in the store except that in c itself the locations are preserved. It is the same location set but the store might be changed because of updations or values. The effect of this is to look at the execution in this updated environment and model it as the eventual effect of the block.

This rule could be applied several times and eventually you will find that when applying this rule you are starting of execution in a store σ L but when you are executing the C in this updated environment your location set could be modified and therefore your store could be different.

So, σ L' is not the same as σ L except that for all locations common to both L and L' they yield the same values. Since we are interested only in the execution of this block with the starting store, we are temporarily updating the environment and the temporary updation of the environment also means a temporary or a permanent change in store but this change in store is just that whatever locations that are not present in the original store might have some values, but all the locations that are common to L and L' should have the same values.

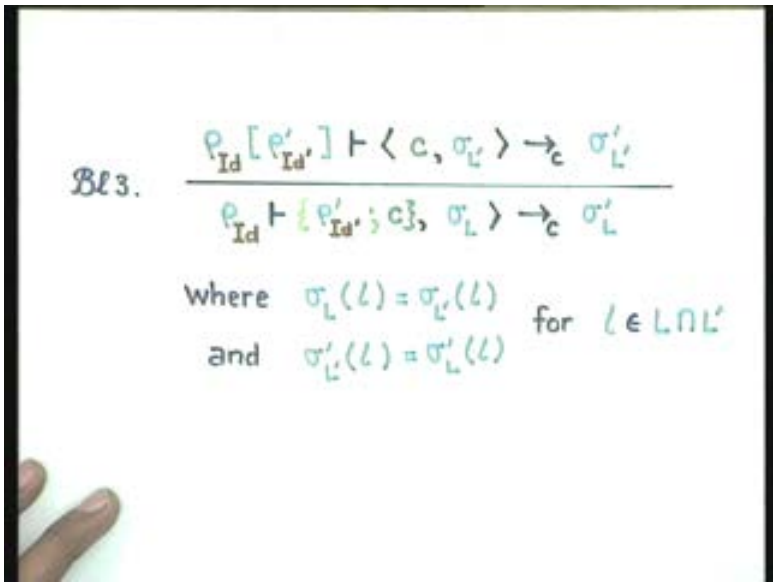
Similarly, in the final store since the creation of the locations L' - L is temporary because of the creation of a little environment which anyway is reversible the store also reverts to where the original set of locations are preserved. So, the irreversible changes in store are anyway subject to a certain scope rule and they have a lifetime. All the locations in L' - L where L' is a larger set than L are really temporary because this environment row prime is temporary. There are two main constraints in all these four stores.

(Refer Slide Time: 15:15)



For all locations that are common to L and L' should have the same value and similarly σ' L and σ' L' should have the same value and lastly, with the same constraints on σ L, once your command in this updated environment finally produces a store of the form σ' L' then the effect of this entire command starting from a given store σ L is to produce a new store σ' L with the same set of locations as it started out with. σ' L is really the same as σ' L' except that all the locations in L' - L have been deleted.

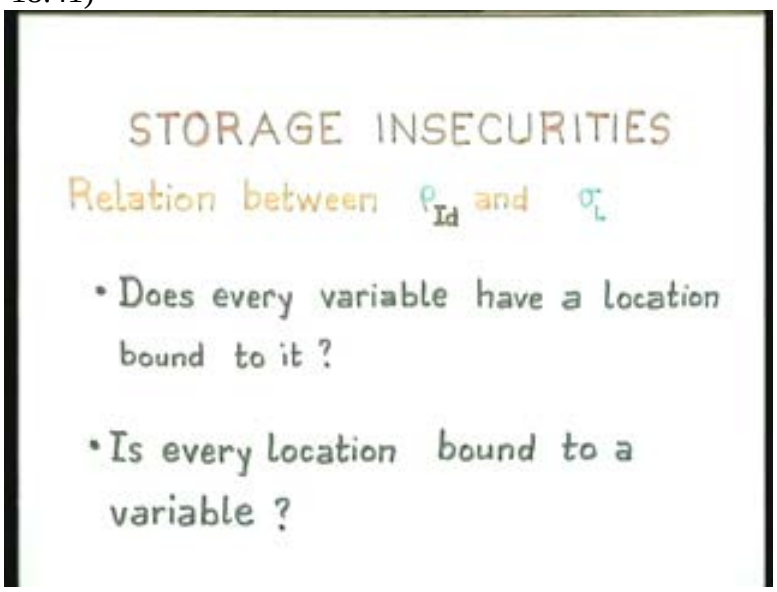
(Refer Slide Time: 16:26)



The fact that it is fairly precise and very concise, it is also dense with information in the block rules. What these rules have shown is that there is some relationship between environments and stores which has to be maintained at runtime. The updations of stores can occur because of two reasons; the creation of new environments or just updation of the current store. It is good to specify various kinds of constraints on what happens if some of these constraints are not met.

If I were to just take an arbitrary environment row on a set of identifiers Id and an arbitrary store σ on a set of locations L then the question is what kinds of relations do they satisfy? Firstly, every variable that is in Id should have a location corresponding to it. Secondly, every location in L is meaningful provided there is some variable associated with it and if there is some variable bounded to it. If there is no variable bounded to it then there is absolutely no way the program can access that location.

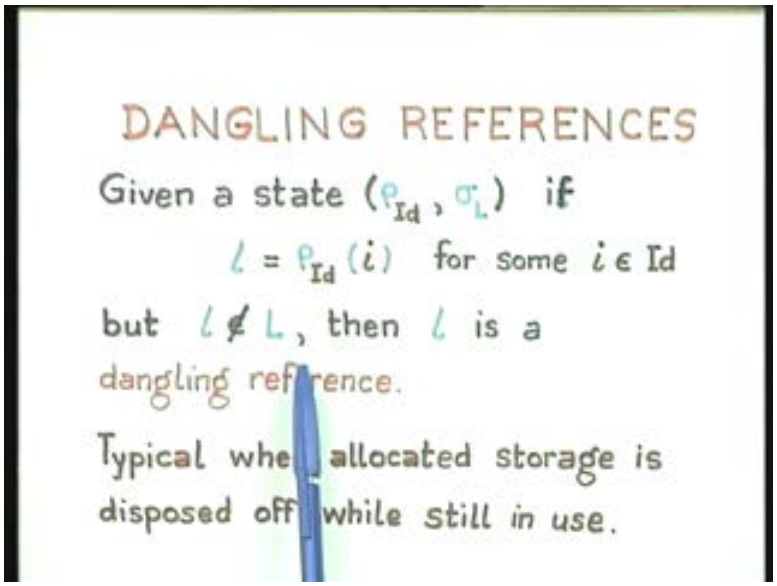
(Refer Slide Time: 18:41)



The main questions are: does every variable have a location bound to it and secondly is every location bound to a variable? With a lot of normal programming especially where storage is dynamically allocated at runtime by a user, one frequent problem is that you get dangling references.

What is a dangling reference? Given a state which consists of an environment row Id and a store σ L, if there is some location L which is bound to an identifier in the environment but that location is not present in the store then there is an identifier which really is not part of the program state. Such a location L is called a dangling reference. This as far as I can see is the most precise and concise way of explaining a dangling reference and this usually happens when you allocate storage to some variable and then you dispose it off while it is still in use and it can be complicated.

(Refer Slide Time: 20:27)



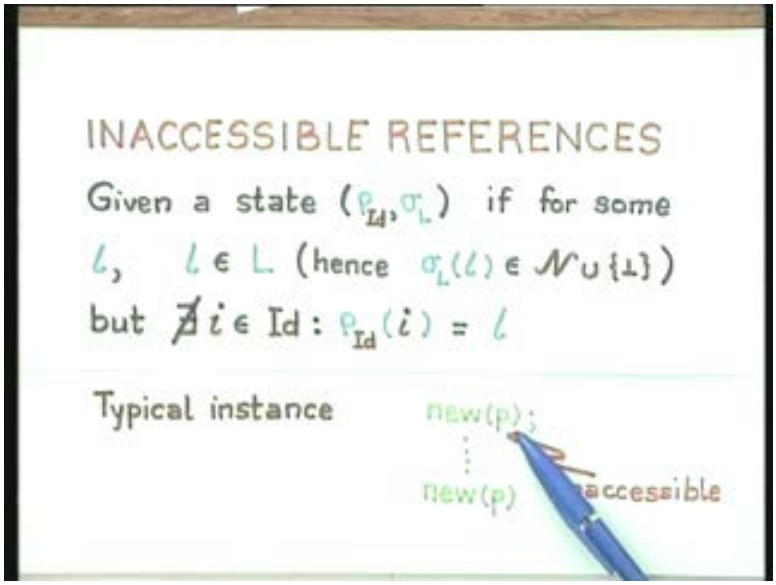
These macros can be significantly complicated when you have aliases for the same location. When a single location is bound to two different identifiers you would say that that location is aliased. The effect of aliasing is that if there are two identifiers P and Q which refer to the same location then you might dispose off P even though Q is still in use in the program and this frequently happens with pointer references in most languages. This is storage insecurity. A simple example is of this form; there are two pointer variables, P and Q, you have allocated a new location to P, you have assigned that P to Q and then you have disposed off P.

(Refer Slide Time: 21:27)

This kind of elementary programming is often caught by the runtime system but there is no reason to suppose that this is the only possibility. You could have very complicated possibilities, very complicated assignments and you might be doing disposals and creating dangling references there by.

Secondly, suppose you have a location which carries a value but that value is inaccessible; typically you are saying that this set of locations L has some location which is not bound to a name. So, if it is not bound to a name then that location is inaccessible. This again happens with dynamic allocation of storage. You do not have any identifier by which you can access that location then you would say that it is an inaccessible reference. The notion of dangling references and inaccessible references and such storage insecurities actually also translate to having a more complicated language and your L values are themselves expressions that could be compound expressions which is when it really makes sense. With the way we have defined our programming language one important property we should be able to prove is that there are no dangling references and there are no inaccessible locations.

(Refer Slide Time: 23:45)



One typical instance again with dynamic allocation is that you might actually be using the same name to create new locations and this creates a new location which is not assigned to anything and this new allocation overrides this previous allocation and therefore the previous allocation becomes inaccessible. It is often quite difficult to debug a program with pointers mainly because the allocation is being done by the user and the user has to be very careful about making sure that there are no inaccessible references or dangling references. The onus is on the user to ensure that he does not create them.

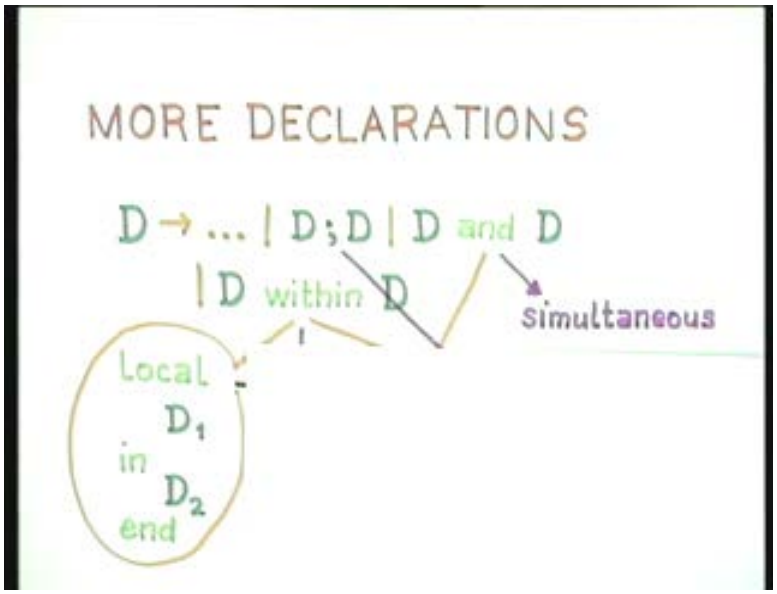
We have actually seen blocks in a command language and the PL 0 compiler calls them blocks except that in the PL 0 compiler every block also has a name. The way we have treated blocks here as unnamed procedures which cannot be called from elsewhere is a simplifying assumption but is very similar to the let construct in ML.

You have some declaration in some expression; Let D in E end. It is very similar to this 'let' construct which we can think of also as a block. It is a functional language and so there are no commands; there are only expressions but these expressions are qualified by declarations or by definitions, as we would call them in a functional language. But we will use the two terms interchangeably. So, our notion has an analog in a language like ML. What it means is that you can qualify expressions with declarations and you can qualify commands also with declarations.

Then the next question is can you qualify declarations with declarations? The only operator on declarations that we have used, if you look at declarations in their pure form, is what might be called the sequential composition of declarations but there is absolutely no reason why, what can be done for expressions and commands cannot also be done for declarations and in fact ML actually has these facilities.

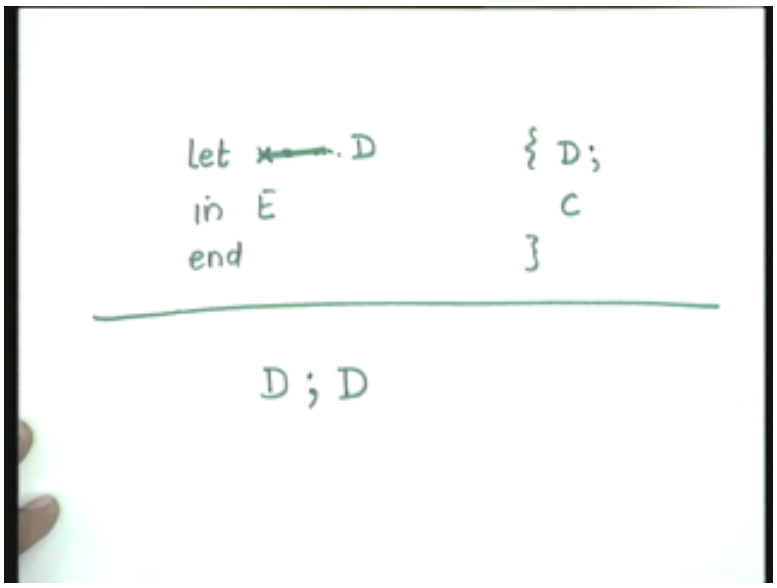
Let us look at some more aspects of declarations and the creation of blocks. I am using a loose syntax for brevity. I do not want it to be as complicated as an ML declaration. We might consider the following kinds of declarations. Most of these imperative languages actually allow only one kind of operator on declarations and that is a sequential composition of declarations. However, ML also has what is known as simultaneous declarations and it has a construct which in order to be consistent with this syntax 'I' uses the word within.

(Refer Slide Time: 28:05)



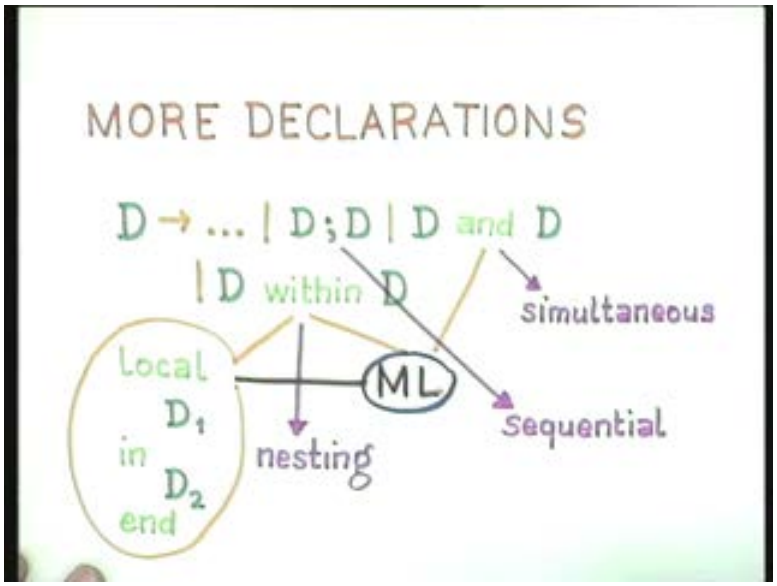
What I write as D1 within D2 would actually be written in ML as local D1 in D2 end. It is different from the 'let' construct. Remember that the 'let' construct has an expression here. It is a grammatically different category. It is also semantically a different category because a 'let' construct denotes a value whereas a local construct in ML actually denotes an environment. So, the two things are different and you reflect their difference by having different syntax too. The reserved word 'Local' in ML is used to distinguish an expression block from what might be called a definition block.

(Refer Slide Time: 29:09)



We will just call this construct ‘within’. It carries this analog of blocks from declarations. So, you can qualify commands with declarations; you can qualify expressions with declarations and create expression blocks or command blocks and you can qualify declarations with declarations and create a declaration block. There is absolutely no reason why we cannot introduce these new declarations into our original language and give semantics because these constructs are completely orthogonal to whatever is there in the command language. You could take that command language and add these kinds of declarations also.

(Refer Slide Time: 30:17)

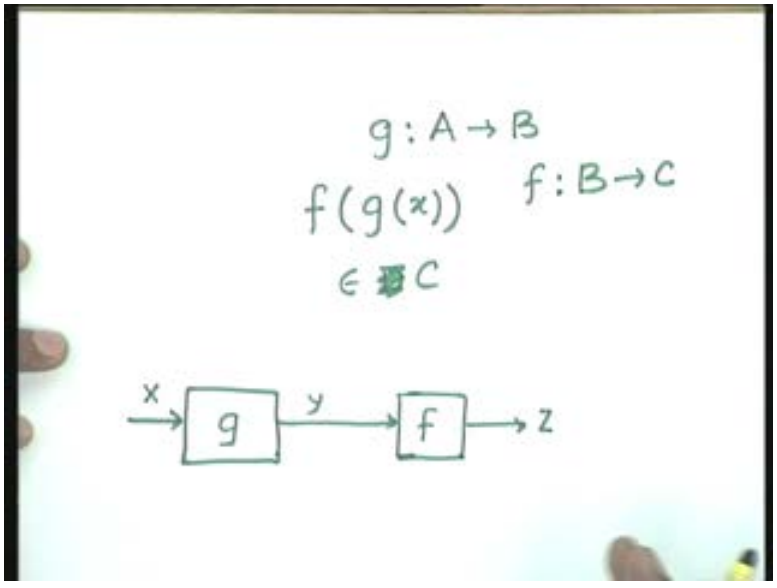


A typical ML program would consist of all possible such declarations. This is of course sequential, this is simultaneous and this is nesting of declarations. This is more like nesting of declaration within an expression and a declaration within a command to create a block. Similarly, you can nest a declaration within a declaration to create a new declaration. We will look upon this in the context of a functional language because then we do not have to worry about store which is just an extra complication. To understand what all of them mean do not really require you to know whether you are dealing with a functional language or an imperative language; you can deal with them independently. Let us look at these three operators in some detail.

If you look at these three operators in detail all these three are what might be called composition operations. The only composition operation that you have learnt are functional compositions in mathematics and thereby also in programming. Sequential composition of declarations is also a composition operation and the other two can also have the same status as composition operations.

Let us look at it in the analog of what we would do with respect to functions. Let us say you have a simple mathematical function and you actually compose two functions; f (g (x)). This f (g (x)) is a simple composition of functions in mathematics where we will assume that the types of f and g are such that you can actually perform these compositions. For example; if g: A→B and f: B→C then this composition of f and g in f ((g(x)) would give you an element of C. This is how simple mathematical composition works and you can diagrammatically look upon this as something in which the value X comes into some black box 'g' probably giving you a value y which is fed into the box f till you get an output z.

(Refer Slide Time: 32:56)



This is how a simple composition works. You can also look upon this as a series connection of these boxes. You can also connect these boxes in parallel just like you do it with resistors. We could actually consider these series and parallel compositions of declarations viewed as boxes.

Given an input environment row the effect of D1; D2 just like the semicolon operation on commands, gives you a sequential evaluation of the two commands. C1; C2 is like putting in the input state into C1, getting an intermediate state out of C1 and putting that into C2 and getting a final state out.

So, it is really a composition operation which is why we will use semicolon as a composition operation rather than as just a terminator. It is not a mere syntactic sentinel to mark the end of a command though more and more programming languages are actually taking the view that sentinels are more important than having it as an operation since most people understand juxtaposition itself as an operation you can eliminate this and use it instead as a terminator. But since we are dealing with composition operations I would like to view it explicitly as a composition operation.

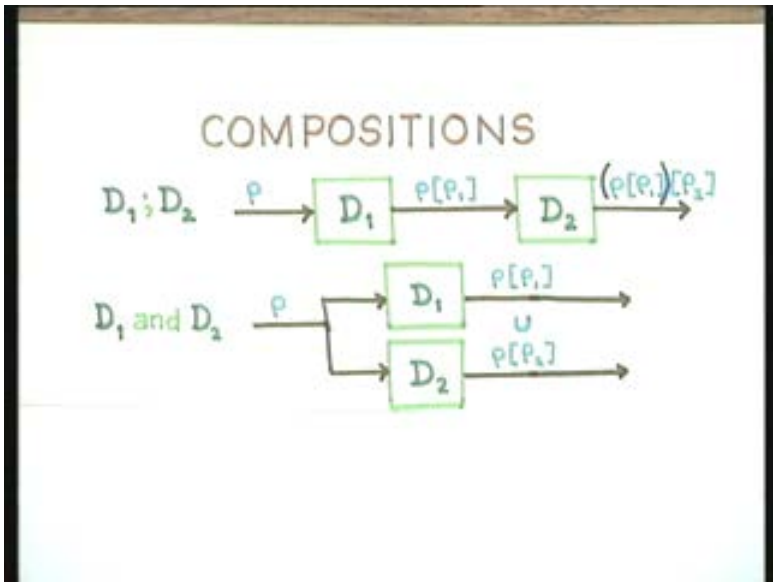
The effect of an environment row on the box D1 is to produce temporarily at least an updated environment. This box D1 actually produces a little environment row1 and all this is supposed to be in the context of some larger command or an expression which temporarily is going to be evaluated in the environment row updated in row 1 and of course the effect of this is to create a new updation where this updation operation is left associate. It implies that you can assume that this bracket is here. Take the updated row, row 1 and update it further with row 2.

If you were to connect things in parallel then what it means is that the same row is applied to actually both D1 and D2 in a fan-out fashion and these D1 and D2 independently produce new environments row1 and row 2 and whatever is the rest of the command or the expression that is to be evaluated is actually evaluated in this composite environment. However, to compose this environment I will just take the union of the two environments. The main restriction that we require is that we should not have the same name declared in both D1 and D2.

If you have the same name declared in both D1 and D2 with different expressions giving them values then of course there is a question of what that identifier denotes in this composed environment. So, that is a syntactic restriction that you have to impose. We could have taken a disjoint union but if you were to take a disjoint union then you have to introduce extra syntax to distinguish between the two identifiers with the same name. It is not a big problem because it is used. The record selector kind of syntax is often used to distinguish different identifiers bound in different declarations.

Very often in languages like ADA you can overwrite the scope rules by using a record selection method to make available an identifier that has been re-declared in the current scope but you do not want it hidden from a previous scope. A simple record selection method can be used to get rid of that but let us not complicate matters. Let us just impose this restriction which actually ML imposes. Unlike ADA for example, ML has this restriction that this D1 and D2 should actually be disjoint in the sense that the identifiers they declare should not have the same name.

(Refer Slide Time: 39:10)

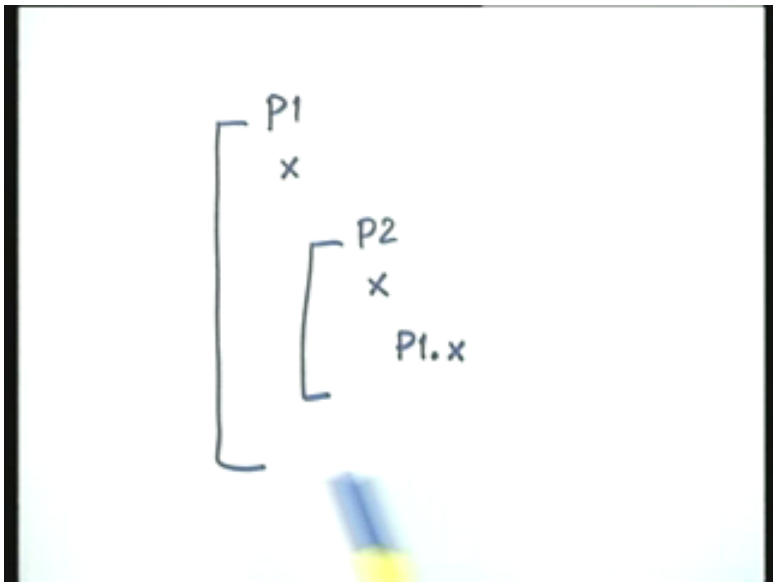


There should be no common names in the two declarations D1 and D2. An objection might be raised like this. Either you take disjoint unions and allow for a new operation called field selection which explicitly tells you whether this y were to disambiguate this y but then in order to be able to do a field selection and explicitly disambiguate this y what it means is that this declaration has to be named and this declaration also has to be named. Unless they are named there is no way a user can actually disambiguate this but once you have named them then you have to have declarations separately as functions. These are complications the Ada manual has gone into great detail to clarify. So, scope rules can be overridden.

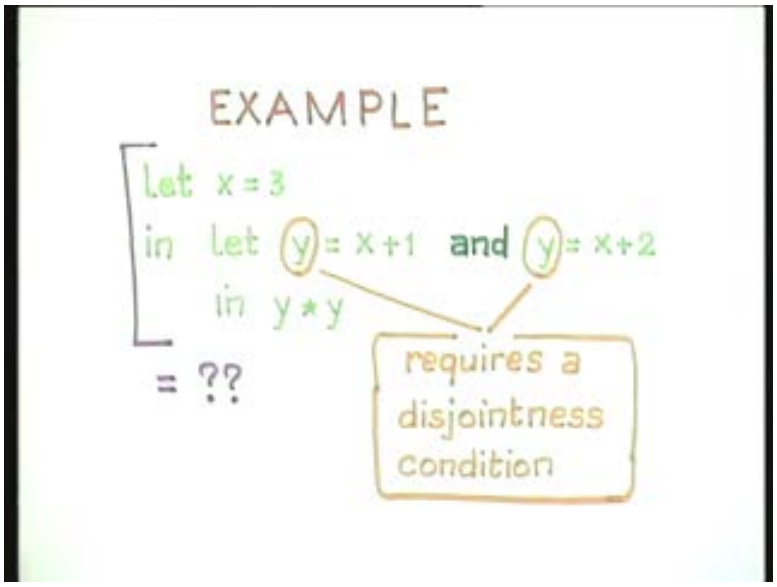
The hole in the scope concept which we spoke about can be overridden by using a record selection kind of operation which basically says that if you have two blocks and if you have an identifier x declared in both blocks and you want to use this x then this block should have a name say ‘P1’. Let me call this P2 and you want to use the x that is in P1 then the Ada syntax actually allows you to do a record selection. The hole in the scope created by this new declaration can be overridden by a record selection mechanism but it assumes that you actually have a name for the block. You have to have a name for the block which means that you are creating a new environment with the name. This name is associated with this entire block. It is an extra complication.

Many people believe that scope hiding can often be used deliberately to ensure that you do not make any confusing assignments. Whereas it is quite possible that one might just use x intending x without actually performing the record selection and one might create an error and it will not be detected. Lots of people believe that as a part of security of programming, the hole in the scope concept is important and should be adhered to and should not be overridden by such mechanisms.

(Refer Slide Time: 42:09)



(Refer Slide Time: 42:30)

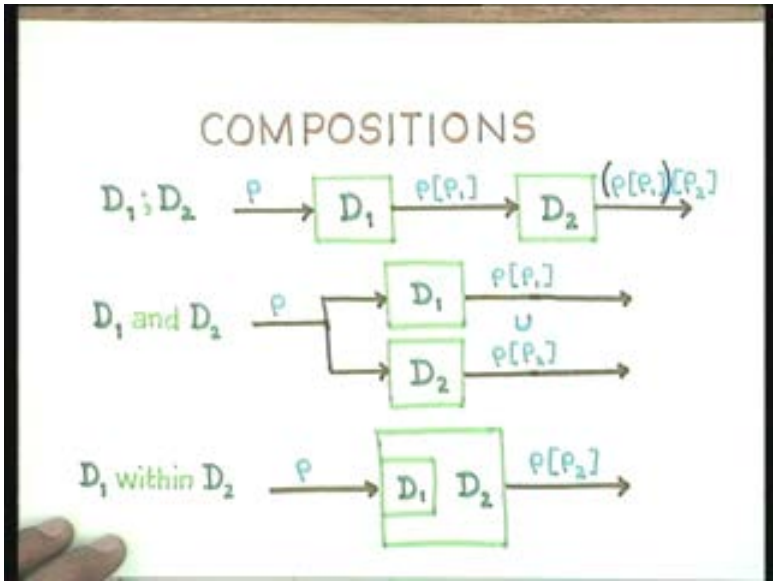


If you try it out in the ML like environment you should get some unresolved syntactic error. It is a matter that can be syntactically resolved. The last composition is somewhat trickier and this does not fully explain what exactly happens. You have a series connection of declarations; you have parallel connections of declarations with the added restriction that the identifiers they declare are disjoint and then you have nested declarations. D1 is nested within a larger block D2. The effect is that the input environment row is really required for D2. However, D2 is so complex that I cannot adequately express the declarations in D2 without introducing fresh new names.

So, I introduce a declaration D1 but which is purely local to D2 and is not visible outside D2.

This purely local D1 actually creates a little new temporary environment row 1 and in this updated environment row, row 1 you elaborate the declarations in D2 and the effect of this is that since this is totally temporary and is bound by a scope which lasts only up to the end of D2 what you get is not something of the form row, row 1 updated with row 2 but what you get is just row updated with row 2. Row 1 was purely intermediate in order to facilitate structuring of the declarations in D2.

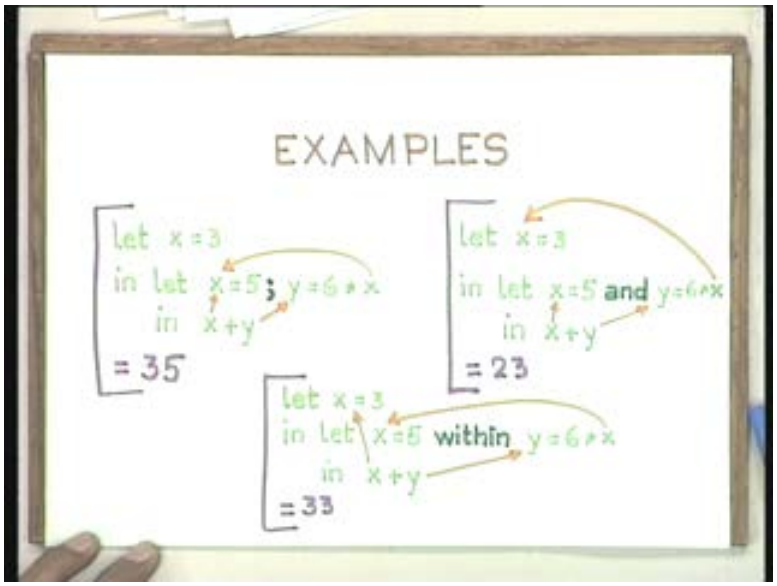
(Refer Slide Time: 44:37)



This is the difference between a pure sequential composition and a nesting of declarations. Let us just look at a simple example to see what this means. I have got three examples one for each kind. Let x = 3 in let x = 5. My ML notation is faulty but that is deliberate so I do not want to be worried about syntax too much and usually there would be Val or Fun reserved words used but I do not want to use all that, so let x=3 in let x=5; y=6 * x in x + y. This is an expression block in which you want the value of x + y where this x according to the normal scope rules refers to a binding created by the innermost enclosing scope which has a declaration for that x and the same is true for this one. These two declarations are sequentially composed.

What it means is that you start with the x = 3. Initially, you have re-declared x to 5. In the updated environment you have x = 5 and in that updated environment y is evaluated. So, for the binding of y the x that is used is really the x in the updated environment where x = 5 and after having created this environment row, row 1 updated with row 2, you evaluate the expression. You would get an answer of 35.

(Refer Slide Time: 48:00)



On the other hand if you had a parallel or a simultaneous declaration then both these declarations are elaborated in the original environment that is created. So, both these declarations are elaborated in an environment in which x = 3; the result of this declaration is to create an updation x = 5 and the result is to create a binding
y = 6 × value of this x which is 18 and this x + y is evaluated in this scope therefore it takes the value of x and the value of y to give you an answer. Supposing you had a ‘within’ construct which is really like a local construct then you have x = 5 within y = 6. So, this declaration of x = 5 is purely local to the elaboration of the declaration. After that the declaration loses its meaning. Then you get the answer.