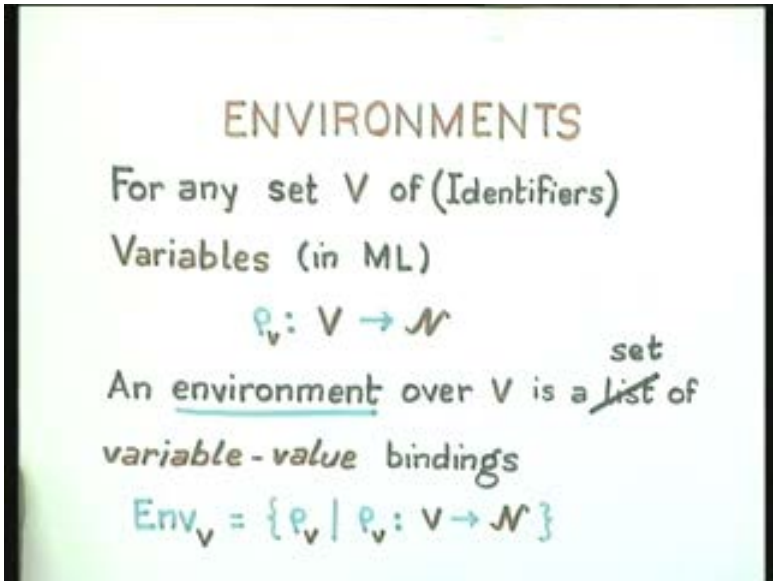
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 16
Lecture Title: Declaration and Commands

Welcome to lecture 16. We will try to integrate declarations within a command language and see how to manage both aspects. Let us go back briefly to what we said about ML like programs.

(Refer Slide Time: 00:40)



Their environments were variable-value bindings and when you have a command language we require something which allows changes to be reversible and something else that allows changes to be irreversible and as a result we require stores in addition to environments.

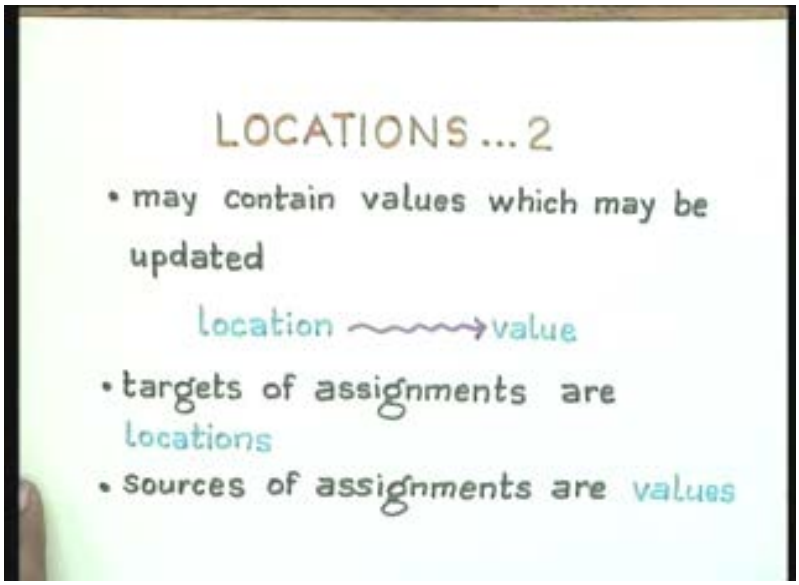
(Refer Slide Time: 00:55)

The relationship between the two will be that the stores will depend upon the actual bindings created by environments and you can only have stores for given environments. As I said there is something implicit about an imperative programming language where the implicit assumption about the existence of memory and the fact that memory can be updated, requires the use of locations.

(Refer Slide Time: 01:35)

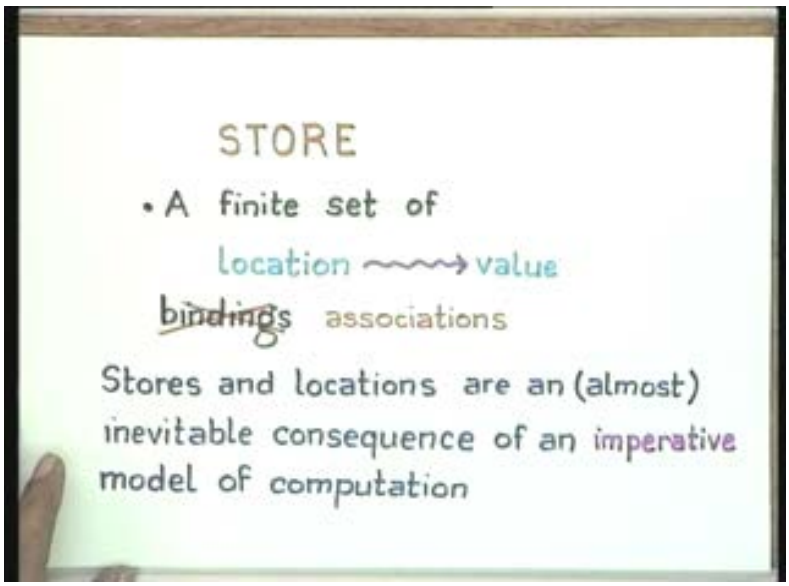
These locations are intermediate between a variable and its value as opposed to constants which would be just identifier-value bindings; variables would be identifier- location bindings and further we would have values associated with locations which loosely speaking refers to values contained in locations. The targets of assignments would be locations and the sources of assignments would be values.

(Refer Slide Time: 02:15)



For any given environment a store would just be a finite set of location-value associations which is an inevitable consequence of imperative language programming. Going quickly through it, we will assume that we have environments created by declarations and stores will be updated by commands. In the case of the most important command that actually allows updation of stores is assignment statement either explicit or in some implicit form, which has a target and a source, where the target actually refers to some location and the source actually refers to a value contained in a location.

(Refer Slide Time: 02:30)



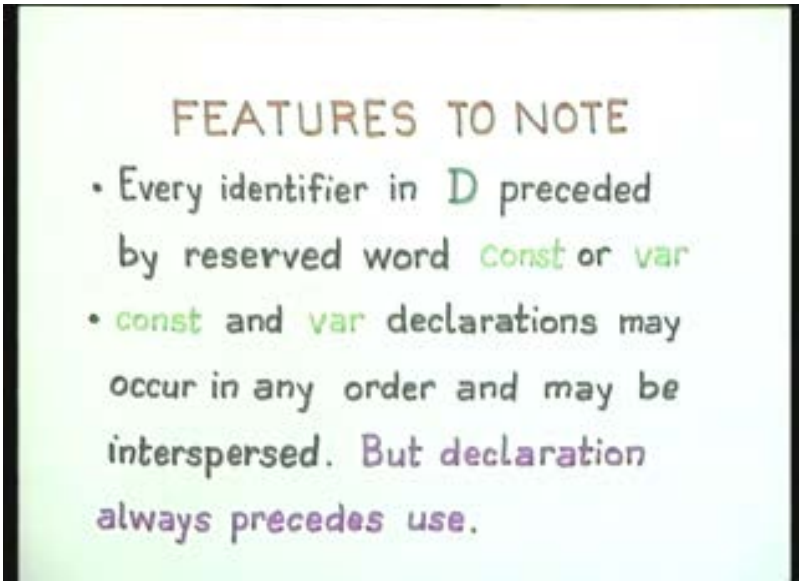
(Refer Slide Time: 03:15)

Even though we use the same name for both depending on the context you will either take the L value or the R value. Let us look at the language. I will just take this very simple language. We have an expression language consisting of numerals, identifiers and binary operations $E -> N / I / E \circ E$. For the present I will assume that these binary operations are fully defined. It is not a partial operation.

We have the usual Booleans and we have declarations. As in the case of the PL 0 language we have constant declarations and we also have variable declarations. In this particular case you can see that since our only data type at the moment are integers it is not necessary to actually define a type here. It is enough to just have a declaration like this and we could sequentially compose declarations, our assignment sequential compositions, conditionals, while etc but more importantly we also have blocks.

Informally, you could look upon this as just declarations followed by commands but this language is not exactly like PL 0 because in this language every identifier in a declaration is preceded by a reserved word ‘constant’ or ‘var’. From the way we have defined the BNF there is nothing which says that all constant declarations should precede all variable declarations. This seems to suggest that you mix variable and constant declarations which do not seriously affect the meaning of the language for example; constant in variable declarations may occur in any order and may be interspersed. But if you follow the principle the declaration always precedes use then it does not really matter because since all the variables are uninitialized in a constant declaration you cannot have any occurrence of a variable declared just before it. The restriction that all constant declarations should precede variable declarations is just a pragmatic restriction and would not really affect much and since variables are uninitialized the constant declarations also do not interfere with the variable declarations.

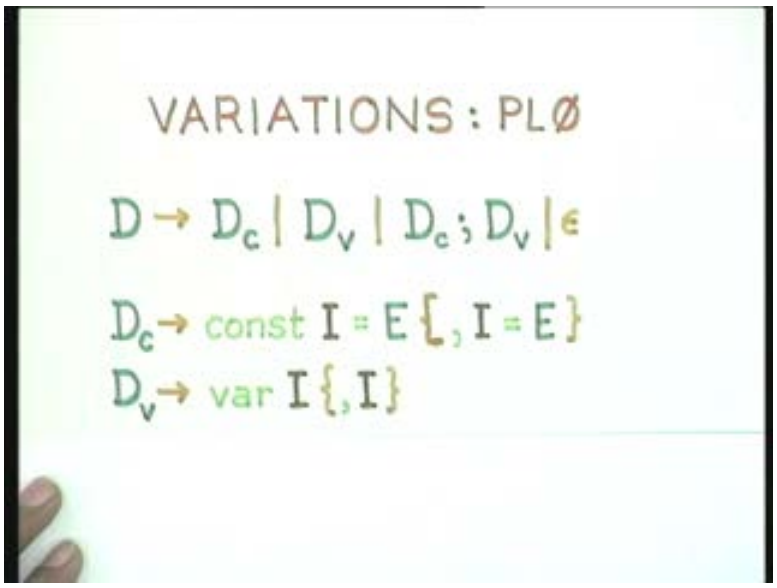
(Refer Slide Time: 07:20)



It really is useful only in grouping together the same kind of declarations into one entity and separating out different kinds of declarations for readability or parsing but otherwise it has no semantic sanctity.

But we will use this basic feature that declaration always precedes use. If you were to actually give a BNF for the PL0 kind of declarations then we would have to have a classification like this where according to those rules you need not have a declaration at all. First of all, you could have just a sequence of constant declarations or just a sequence of variable declarations or a sequence of constant declarations followed by variable declarations. I have introduced two new non terminal symbols Dc and Dv for constant declarations and variable declarations and the PL 0 kind of constant declarations and the variable declarations could be written out in extended BNF language.

(Refer Slide Time: 08:54)

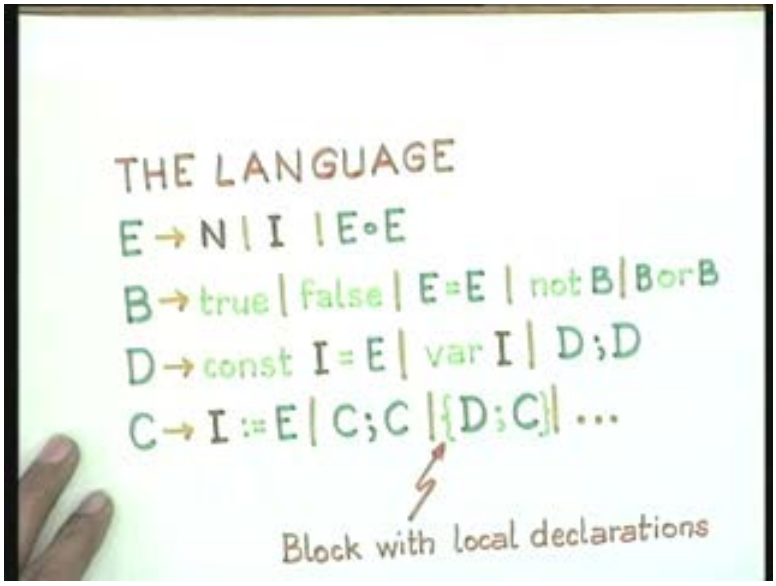


If the reserve word constant occurs then there has to be at least one constant declaration followed by 0 or more occurrences of other constant declarations separated by commas and similarly with variable declarations.

Of course you do not need to have any declaration at all. As far as blocks are concerned it is important that they are really like unnamed or anonymous procedures and as a result the question of naming is actually very important in any kind of programming. If you have an unnamed block then it cannot be called from anywhere which means that it is a purely local block and therefore if there were any declarations they are purely local to that part of the block.

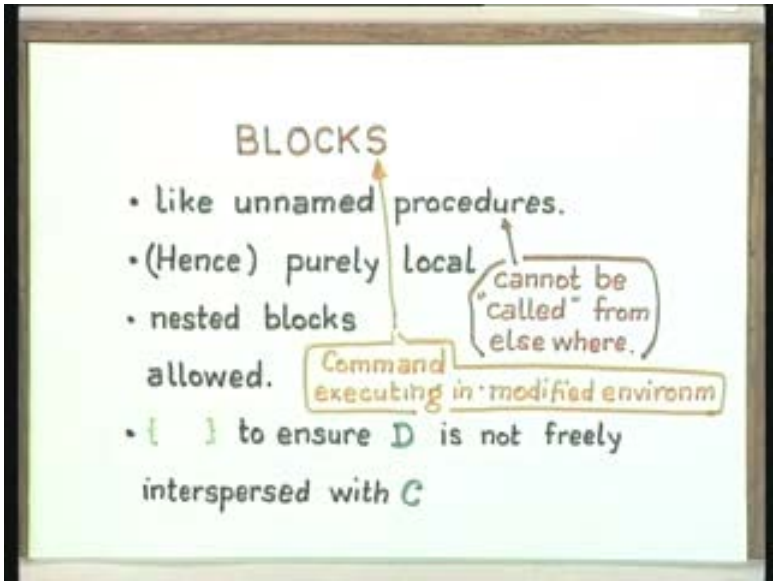
In the BNF I have put in this pair of braces here to emphasize the fact that supposing you intersperse declarations and commands then you cannot arbitrarily intersperse them because declarations create environments, little environments and commands update stores given the current environments. These braces indicate actually the scope of a declaration. So, the whole point is at issuing the arbitrarily mixing of declarations and commands.

(Refer Slide Time: 11:15)



If you did arbitrarily mix up declarations and commands such that a declaration is available throughout that command and you did not have these braces then that is equivalent to moving all the declarations up to one place and separating out the declarations from the commands, which would be a cleaner design. The idea is that if you were to actually intersperse declarations with commands then you really want to have some extra local declarations and therefore you want to create a new block not otherwise. This means that you have to clearly delimit the scope of those declarations that you intersperse. This is really like an unnamed procedure. The declarations are purely local to the scope within which they occur and they cannot be called from elsewhere. You can nest blocks in this fashion because a block is just a command. The braces act like a ‘begin’ and an ‘end’ and so these blocks are really like Algol 60 blocks and you could have local declarations within a ‘begin’ and an ‘end’.

(Refer Slide Time: 12:46)



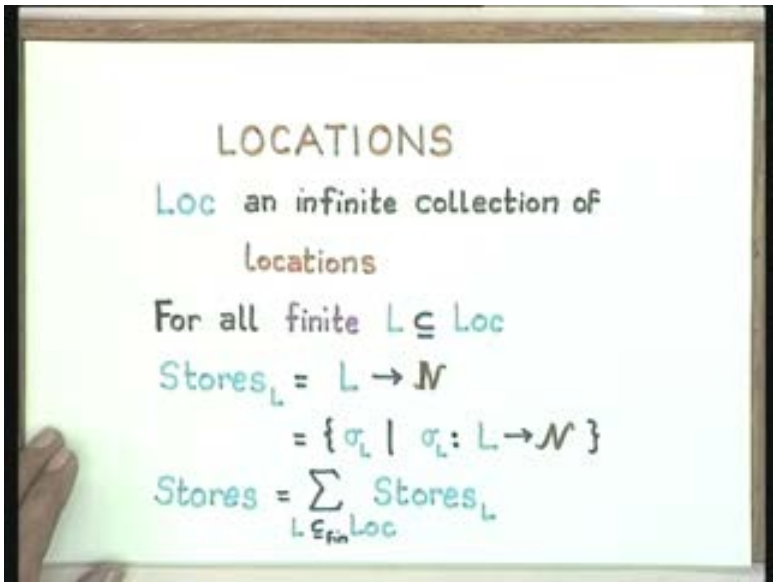
You could nest blocks and these braces really have no role to play except that they clearly delimit the scope of the declarations contained within them. Now let us look at the semantic definitions in such a setting.

Let us look at the notion of locations. In any kind of semantic definition we of course assume that the meaning of the language is really independent on the bounds of resources that are available for the implementation of the language. We assume that there is an infinite collection of locations and these locations may all contain some values. However, at any time in the execution of a program at any instant in its runtime behavior you require only a finite number of locations. There are only going to be a finite number of locations used by a program. So, what should be available is an inexhaustible supply of memory but at any instant of time in the execution of a program only a finite amount of memory is actually being used. The amount might dynamically vary but it is always only a finite amount. For any finite set of locations, we will define stores on that set to be the collection of all location-value associations.

Given a finite set L, the set of all possible values that may be associated with each location is a store at any instant and the set of all stores is a set of all such possible mappings that you can draw and these stores are of course parameterized by the finite set of locations that you are

considering at the moment. As usual the set of all possible stores is just a disjoint union of the set of stores for each finite subset of locations. So, this is our notion of locations and this is our notion of stores. Let us then look at the notion of environments.

(Refer Slide Time: 16:00)



Our language is such that it has more constant declarations and variable declarations and the constant declarations in the language are exactly like the variable declarations in an ML like functional language. A variable in a functional language really denotes the name of a value that remains unchanged throughout a scope and that property is fulfilled exactly by our constant declarations.

The variable declarations are the ones that change. Part of the reason why there are only a finite number of locations used by a program is that it starts initially with only a finite number of identifiers and therefore requires only a finite number of locations initially and on demand more locations might be required but unless it is an infinitely executing program always there will be only a finite number of locations used.

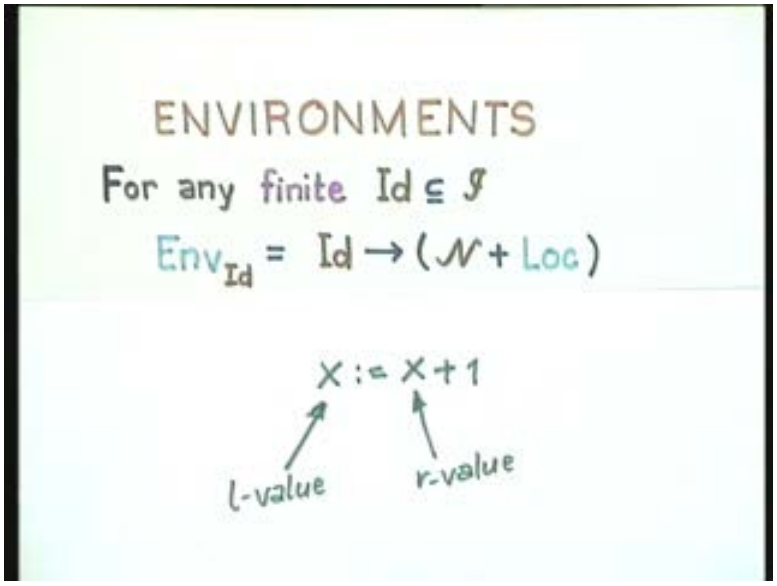
Given a finite collection of identifiers, an environment is just a binding from identifiers to values or locations and of course this '+' here is a disjoint union which means that at any point for any identifier in an environment it is possible to determine whether it is a location or a value.

Remember what I had said about disjoint union. If you take the disjoint union of two sets, firstly; if they are two finite sets then the disjoint union contains as many elements as the sum of the two sets and each element also maintains an identity in the component sets from which it is actually taken. That is a disjoint union and the set of all environments given a collection of identifiers. What I said last time was that there seems to be some confusion between the L value and R value. What I said now was that the same name is being used for both to denote both L values and R values and the context determines what we should take. It is not that you are allowed to take anything. What this further says is that your R values could also be locations. So, the matter is quite complex; it is not so simple. The fact that locations could also be used as storable values means that locations could be R values themselves of certain expressions and you require a dereferencing facility to find the R value of another R value which is actually an L value of some other expression.

In the dereferencing operations in most languages like C, you have a star which does dereferencing or you have the circumflex in PASCAL which does dereferencing. That only says that you can have an expression whose R value is actually a location and if it is a location then it means that you can actually locate its contents. You can take the R value of the R value of an expression provided you ensure that you are taking things in a disjoint union fashion so that the identities are maintained. For example; you cannot look at the contents of a value of this form but certainly you can look at the contents of a value of this form and if these are going to be the storable values then this process can go on ad infinitum. All that I am saying is that the same notation is being used in most programming languages and the context determines whether it is an L value or an R value.

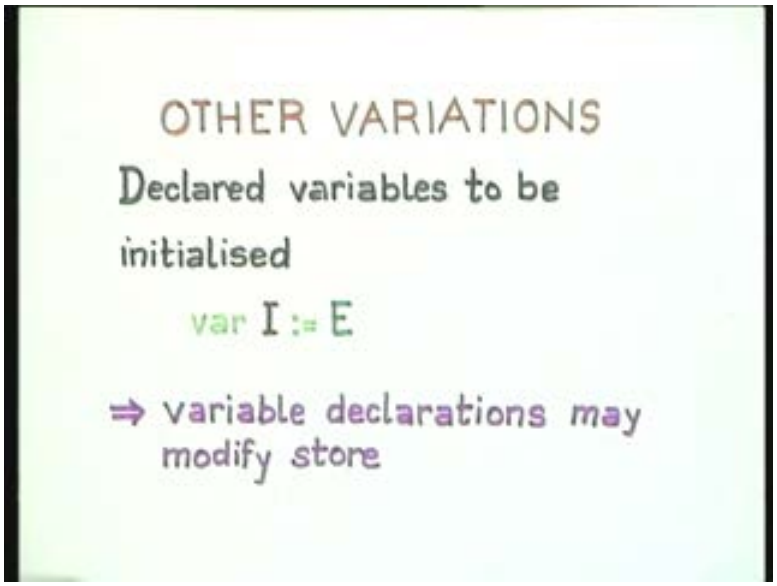
Secondly, R values could themselves be L values of some other objects. For the present let us not get into the complexities of L values and R values. We will just look at the simplest form which occurs when you have a statement like this; for an identifier ‘ x;= x +1’, x denotes the L value which means the identifier- location binding. In ‘x+1’ x denotes the R value which means the value contained in the location to which the identifier x is bound and the context here determines when you should take an L value and when you should take an R value.

(Refer Slide Time: 22:22)



The context tells you if it occurs on the left hand side of the assignment then you are talking about a location and if it occurs on the right hand side then you are actually looking at the value inside a location. The set of all environments is just again the disjoint summations of all possible kinds of environments. For example, many other languages allow declared variables to be initialized in the declaration itself.

(Refer Slide Time: 23:36)



This is to prevent uninitialized variables from being used or accessed for example. However, this is not always possible, for instance if you are declaring a large array in a language like ADA then clearly it is not possible to initialize it at compile time. It is simpler to initialize the array by reading the values from a file through an explicit initialization rather than trying to initialize it at declaration time itself. This means that variable declarations could in general modify the store.

Now as you elaborate a declaration your environment changes so the associated store also changes. In fact you are moving from some stores L to another stores L prime as you elaborate a declaration. Further even though your variables are uninitialized we have to really talk about uninitialized variables as actually containing some value. After all if your store is a function from locations to values, the moment a location has been created that means there is a value contained in it. Otherwise your store's function is not really a function. Let us impose one new condition which is that we will add a new value to the set N and this value really denotes some undefined value and what happens with this value is that since it is just a value we really have to look at its effect on the expression language. Suppose this is an uninitialized variable in some expression, what is the value of that expression? Most languages would just tell you that this is not absolutely essential.

For example; languages like FORTRAN which do not require declarations of variables actually take an initial value, if it is an integer variable, to be 0 or if it is a real variable to be '0.0'. So, if you have not initialized a variable then very often it is assumed that it has an initial value of 0. It is like an implicit initialization of the variable the moment it is encountered. However, that often complicates matters both pragmatically and semantically because firstly if I have not initialized a variable why should you assume that its value is 0 or its value is some identity element which is the first point.

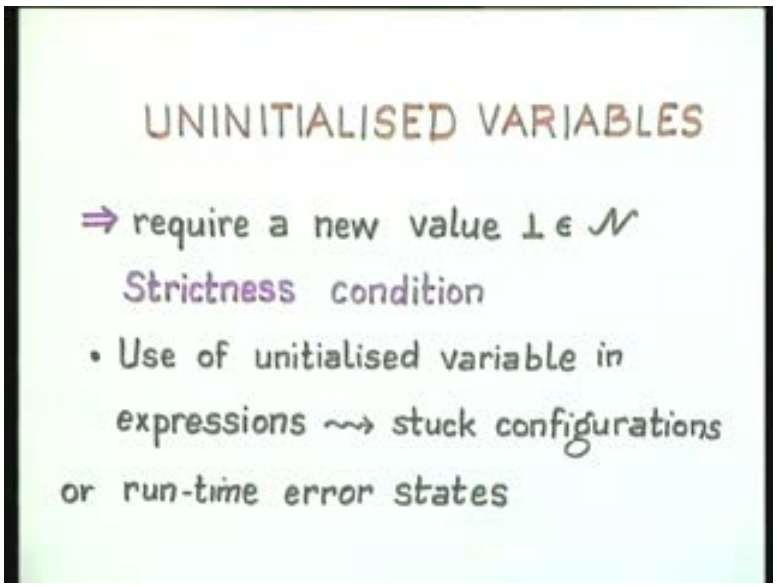
The second point is that even though your semantics assumes that there are an unbounded number of locations, pragmatically you are going to reuse memory as and when it gets de-allocated and the house keeping is not going to be done in memory that is de-allocated which means that that area of memory could have previously contained some values when it was previously used and if you did not go through a fresh initialization the old value might get used.

So, an uninitialized variable is potentially hazardous. It is safer to actually assign it a value that is clearly different from any of the workable values that you have because if it already contains some value without your knowledge and it is actually being used, then it does not differentiate between an error of judgment on your part and a deliberate attempt not to initialize it because you think that the initialization would be done automatically.

We will take the safe view that there is a new value which is there in the underlying virtual machine but it is not part of the language; it is not realizable in the language and a programmer cannot explicitly use it. It is only in order to provide a runtime or compile time error checking for uninitialized peripherals. When you adopt this approach then all expressions which use uninitialized variables become meaningless. The values associated with all expressions which use uninitialized variables are such that either they become stuck configurations or they lead to runtime errors.

This is what happens typically in PASCAL. Very often even though it is easy to do, most PASCAL compilers do not actually check whether your variables are all initialized at compile time.

(Refer Slide Time: 29:40)

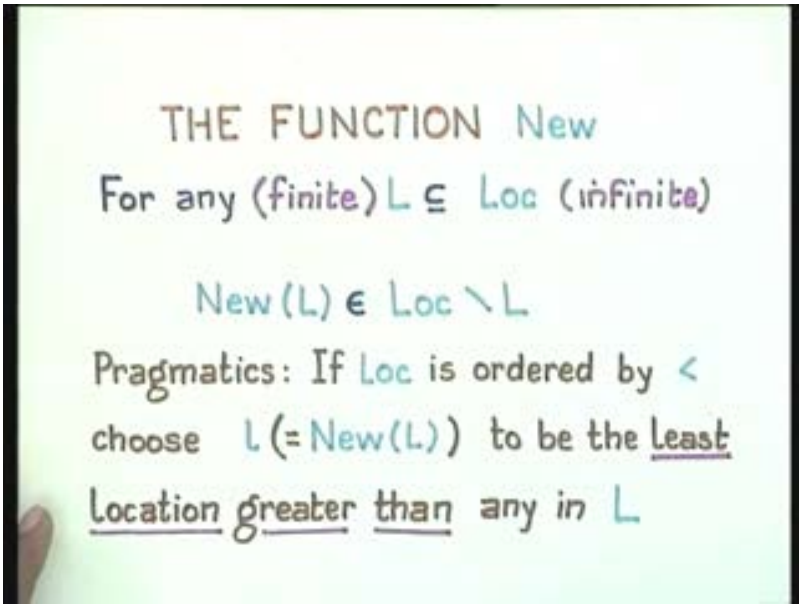


However, at runtime they do point out that there was a variable without a value and what that value is, is some kind of a null value which is assumed initially to be stored. During allocation that PASCAL runtime system actually cleans up that store so that it does not contain some previous values but this is not implemented properly in many compilers and very often you will find that the previous values that the location had would be used sometimes without any warning or an error message.

The PASCAL language is clear in that the uninitialized variables should be explicitly pointed out and should give a runtime error. Before we get on to the actual semantics the assumption of an unbounded number of locations means that every time a new variable is declared you should get a new location. If at any stage you have only a finite number of locations in use then it is always possible to get a new location.

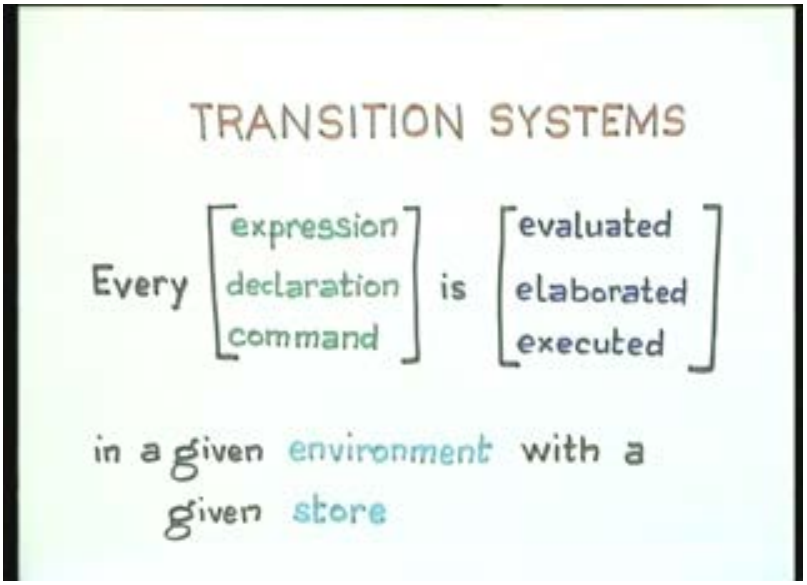
We will define an implicit function for this purpose which I will call new when given any finite set of locations it gives me a new location that is unused. Pragmatically, if you can assume that the locations are all ordered by some ordering then you can assume that the new of L gives you a location with the least location greater than whatever exists in L for example. We will assume the existence of this function which we will use as a site condition. This function is absolutely essential because it is also useful to do dynamic allocation of memory for example, when you have pointed declarations then you actually explicitly allocate and de-allocate. It is also useful for implicit allocations and de-allocations. You could correspondingly define a delete but then that is not essential in a semantic model which assumes an unbounded number of locations but delete is essential when you have a finite number of locations so that reuse is possible.

(Refer Slide Time: 32:58)



We will look at the transition system. From now on we will not look at the complete transition system. We will look at only the modifications that have to be made to the transition system so as to be brief and clear about exactly what we are doing. So, every expression is evaluated in a given environment with the given store. Both the environment and store are parameters of our transition tools for all possible categories.

(Refer Slide Time: 34:03)



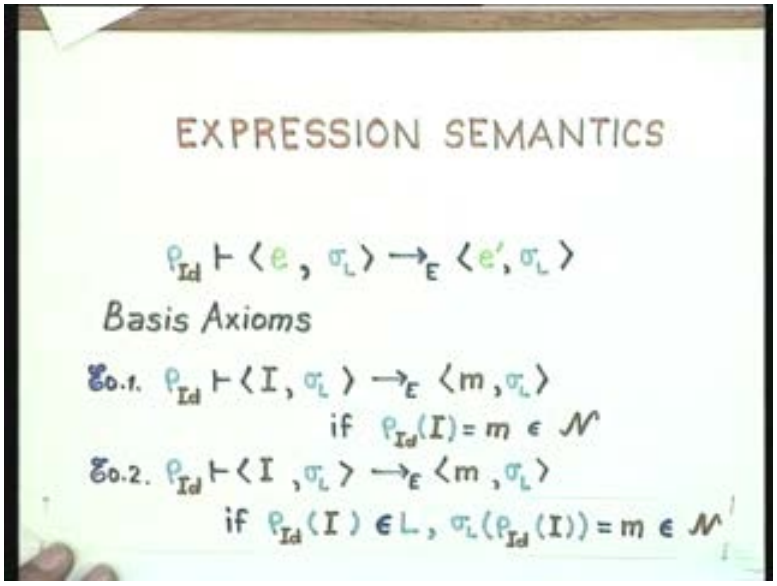
Our typical expression transition system for example will look like this. We will combine the two concepts previously. We will look upon any expression as being evaluated in this environment with this store. Now most of the expression semantics can be rewritten in this form without much problem.

The major change is that now you have two kinds of identifiers. They are either constant identifiers or variable identifiers otherwise in our case the expressions can occur on the right hand side of assignment statements. That is really the only kind of expressions that you need to worry about. So, the expressions that occur on the right hand side of assignment statements will all require their R values to be used. Only variable identifiers can occur on the left hand side of an assignment statement in the case of this language.

Let us just look at the expression semantics in terms of just these two distinctions. The expression axiom e NOT is actually going to be split into two. I make it e NOT 1 and e NOT 2. The value of this identifier is just a natural number m, provided the environment declares it to be a constant identifier. If the environmental binding for that identifier is the constant m, note that this m actually could be this undefined value 2.
It will not be this undefined value in a constant declaration because that undefined value is not explicitly available to the programmer except by certain devious mechanisms like dividing some two previously declared constants by 0. So, it is realizable in some strange ways but it is not explicitly available to the user. This m could be an undefined value. This tells you that if the environment associates with this identifier; note that we defined our environment as a mapping from identifiers to the disjoint union of values and locations. This means that with every identifier is also associated its identity as to whether it denotes a value or a location. If the environment associates a value with it then the meaning of this expression is just the corresponding value in that environment. On the other hand if the environment associates a location with it then the meaning of this expression is that firstly the environment has to give the identifier a location binding and if it gives it a location binding then it is possible to look at the value inside that location and take that value. In either case for an identifier that occurs just as an expression, which means that here I am assuming that within this language it only occurs as an

expression on the right hand side of assignment statements, you also have a corresponding value but there is a level of indirection if it is a variable, which is not there, if it is a constant.

(Refer Slide Time: 39:05)



The expression language as far as we are concerned does not contain any L values in this simple language. It contains only R values. Therefore we have only these two cases to really consider. All other expression semantics will be more or less whatever we have done before. There is nothing much to the expression language however we have to look at variable declarations. Firstly, there is no fuss to be made about constant declarations because the constant declarations are exactly like the ML variable declarations.

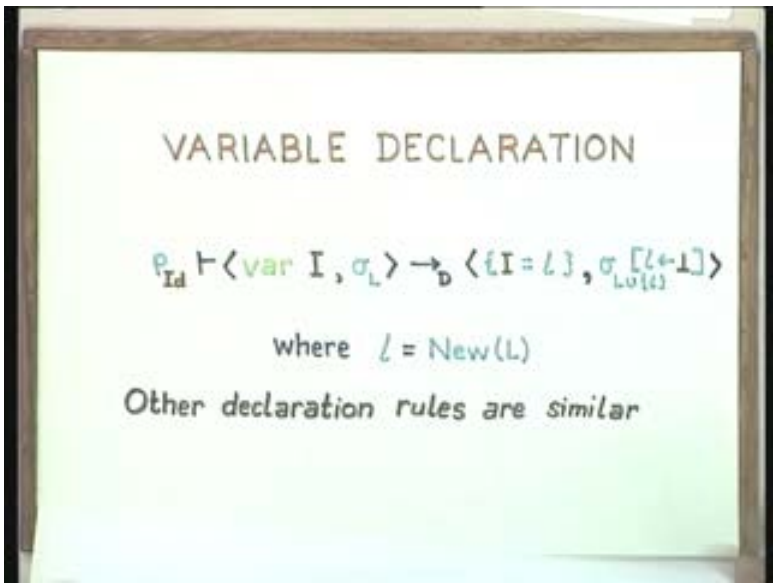
In an environment a constant declaration of the form I = e means that you evaluate the expression e and then you create a little environment in which I is bound to that value in that state. There are no side effects in this language, so, there are no other changes to be concerned about. However, in the case of a variable declaration firstly what it means is that given a store sigma based on the set of locations L; given an environment already which has a collection id of identifiers, when you encounter a new identifier or an old identifier which is already present in this, then firstly you acquire a ‘New’ location and this ‘New’ is very important in the sense that you are acquiring a location that has so far never been used. There is no confusion between this

location and any other location that is present in L. This l has to be different from anything else that is present in the capital L.

So, you first acquire a new location and bind that identifier to that location but that location has to have a value so that your store gets altered. Your store instead of being a function from capital L to values becomes a function from capital L union this new small L to values and you have to associate a value with the identifier. Since it is an uninitialized variable you associate the undefined value with it. As part of a variable elaboration your store actually changes. The store does not really get updated in the sense in which we normally talk about store updation. But this store gets extended to a new store where all the values of the old store are preserved plus a new location has been added and therefore a new value to that location. The other declaration rules are similar so you can take the sequential composition of declarations and create new environment store pairs.

Since the environment has got changed and the environment introduces new location the store also gets changed in a certain fashion.

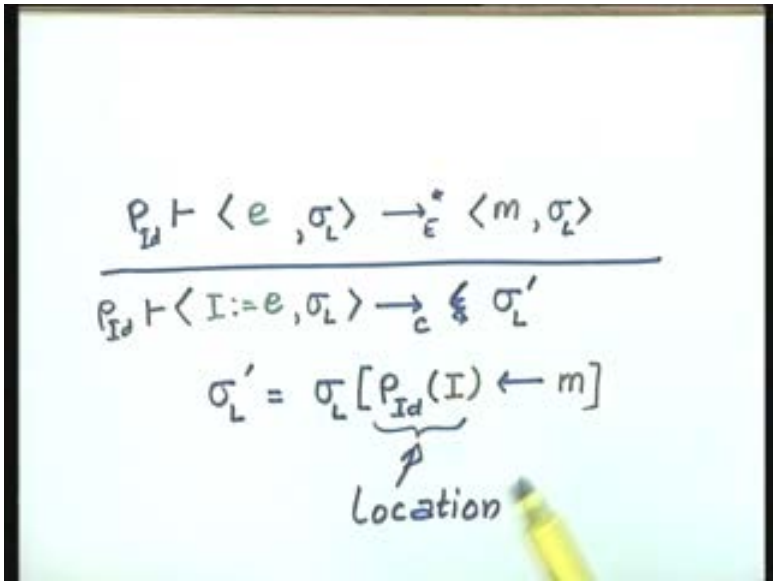
(Refer Slide Time: 43:19)



I will assume that you can take the assignment statement semantics as before except that now you have to take the assignment statement semantics as being of the form in a given environment

row, if in a given store sigma you have an expression e which eventually goes to a value m and since there are no side effects created by expressions we can consider the same L and id here, then the effect of an assignment of the form I is assigned e in this state σ L to create a new store sigma (L') where σ L '=σ L with the updation that the location of the identifier I has associated with it the value m. This is actually a location and in the case of an assignment this is clearly an L value. You take the L value of the identifier but throughout the expression you are only taking R values and evaluating the expression in this store and this environment to eventually yield the value m and associate that value with it.

(Refer Slide Time: 46:40)



That is the only change in the assignment statement. All other commands remain more or less the same except that we should look at the block commands which have created new declarations.