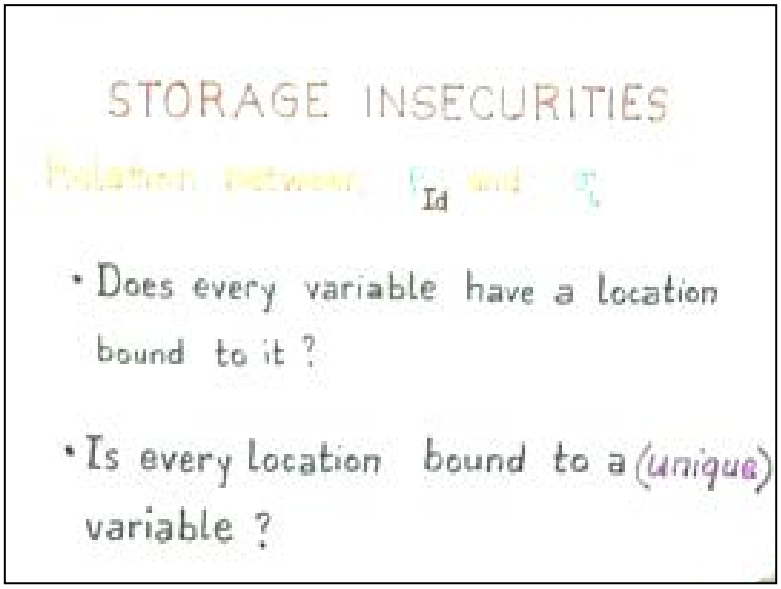
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

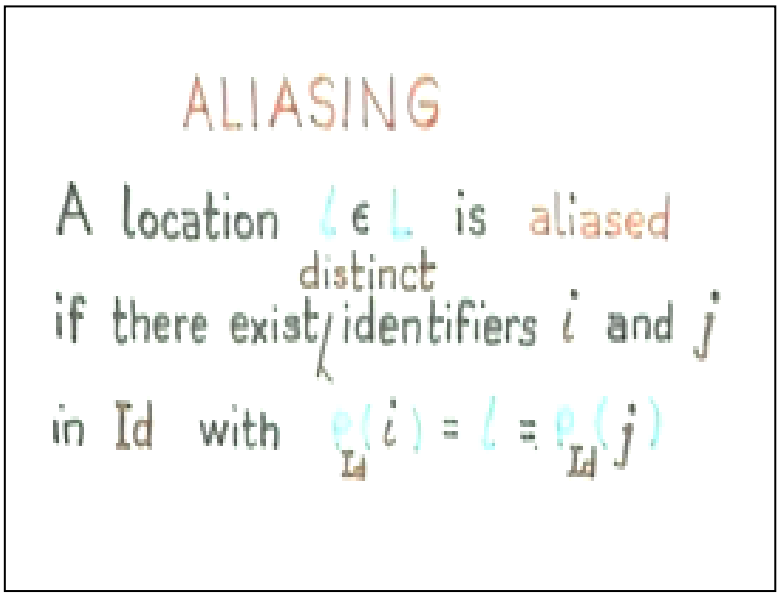
Lecture no 18
Lecture Title: Qualification

Welcome to lecture 18. We will briefly summarize what we did last time and go on to the principle of qualification. Last time we looked at the relationships between environments and stores. Given an environment row on a set of identifiers Id and a store on a set of locations L one question that arises is: does every variable have a location bound to it? Then secondly is every location bound to a variable and in particular is every location bound to a unique variable?

[Refer Slide Time: 00:47]

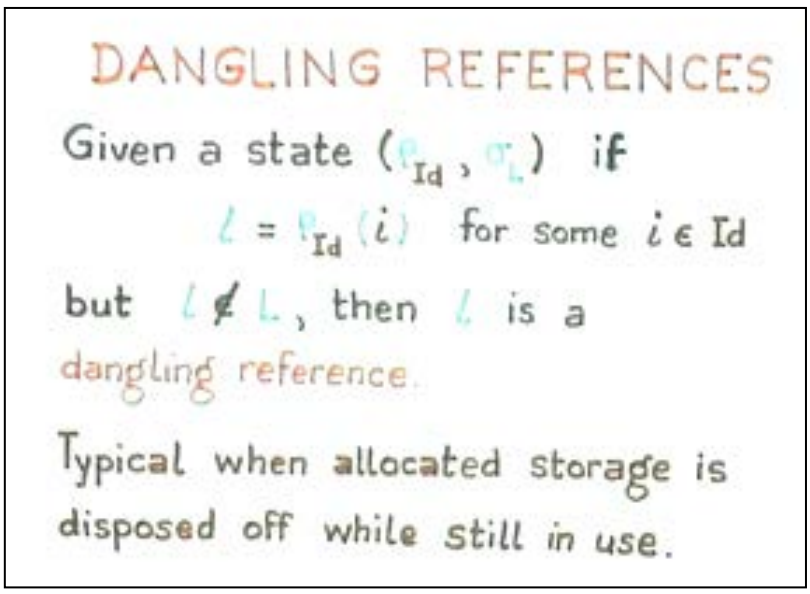


[Refer Slide Time: 01:18]

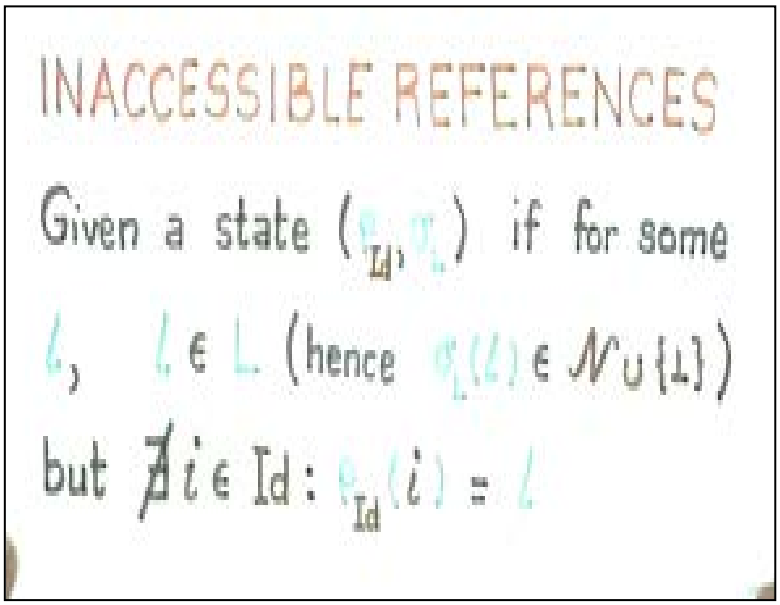


Firstly, there is a question of what is known as aliasing. A location L is aliased if there exists distinct identifiers i and j that refer to the same location. Aliasing and dynamic storage allocation can lead to these problems like dangling references where L is a dangling reference if there exists a variable which refers to L but there is no corresponding location in the set of locations.

[Refer Slide Time: 01:34]

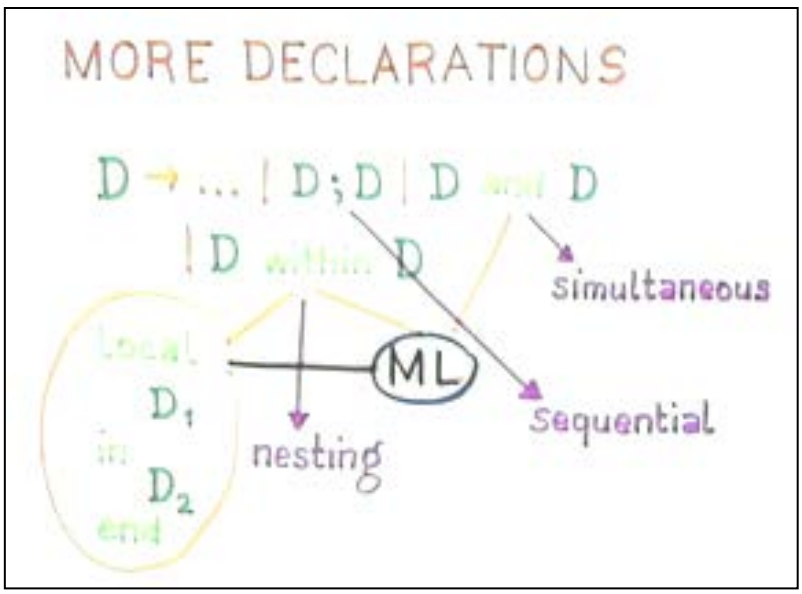


[Refer Slide Time: 02:02]



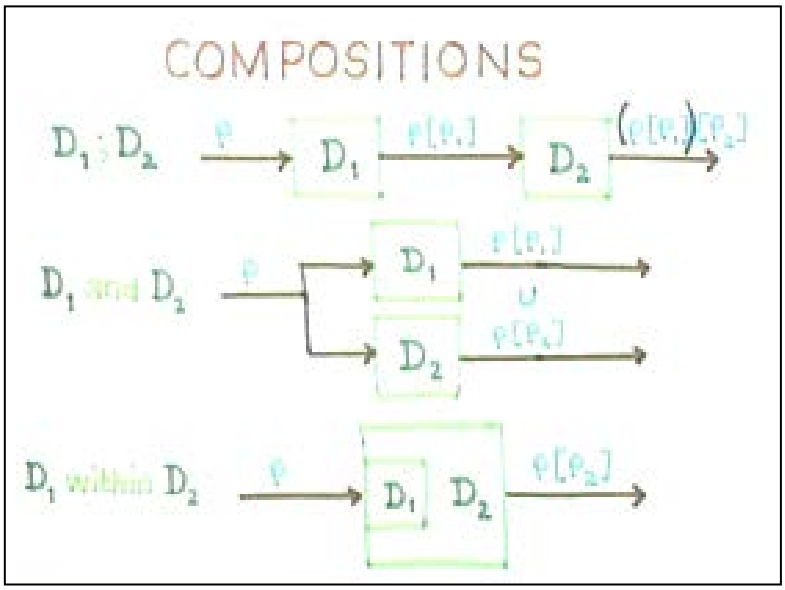
Similarly, you can have inaccessible references. If you have a location in L and there is no way of referring to it, you would say that the reference is inaccessible. Then we also looked at some more forms of declarations. As I said we could think of various composition operations on declarations.

[Refer Slide Time: 02:18]



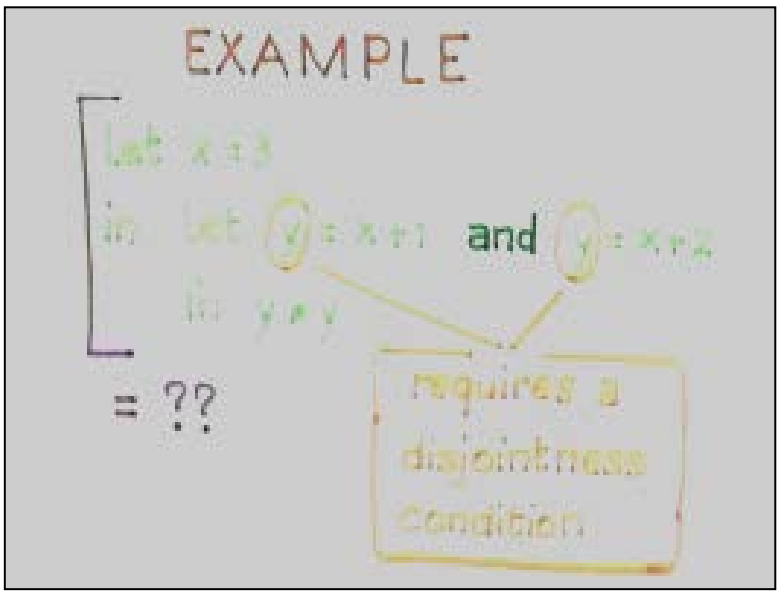
Let us look at some M L type declarations. The main composition operations in M L type declarations are that you could sequentially compose declarations; you could have a simultaneous composition of declarations or you could have nested declarations. These are really the main kinds of compositions one can normally think of on structures.

[Refer Slide Time: 02:56]



The three kinds of compositions have approximately the following kind of diagrammatic representation. You can think of a sequential composition as taking an input of an environment, creating a new updated environment in which the next declaration is to be elaborated and that in turn creates a new environment. In the case of the simultaneous declarations you elaborate the two declarations in parallel. In other words you elaborate them both in the original environment that you have and you take the union of the two environments that are produced. In the case of D 1 within D 2 that is in the case of nesting, D 1 is a sort of via media by which the main declaration D2 has to be elaborated producing a new environment row 2.

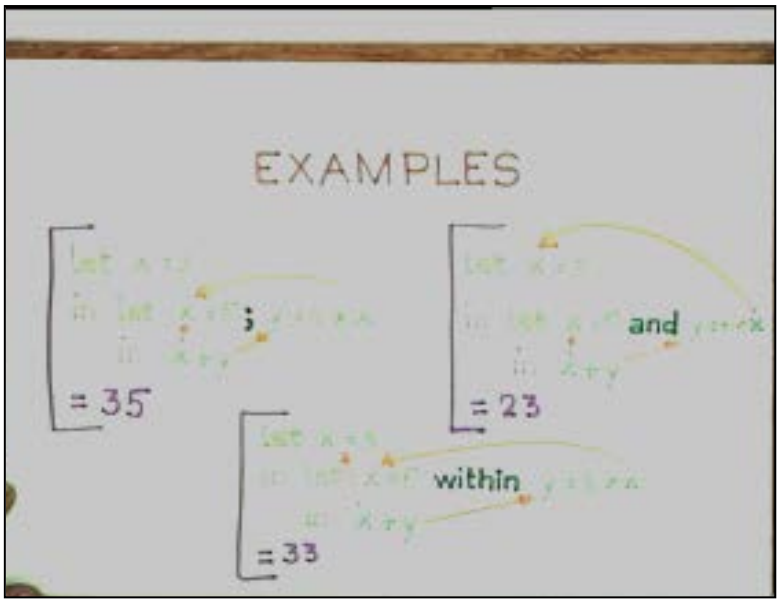
[Refer Slide Time: 04:02]



The simultaneous composition actually requires a disjointness condition so that you get a final environment in which the two environments are disjoint and there is no

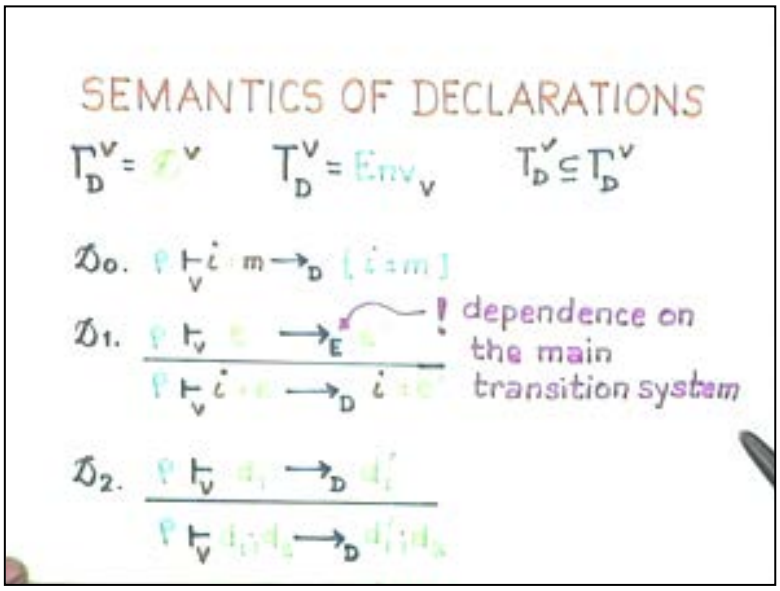
confusion. The three kinds of declarations actually yield different semantics as this example shows. In each case you have to think about where exactly the binding is and what a reference of each identifier is. In the case of a sequential composition the elaboration of this declaration acts as an input to the following declaration and therefore any reference to x here refers to the most recent occurrence. Here x of course also refers to the most recent binding and you get the answer.

[Refer Slide Time: 04:30]



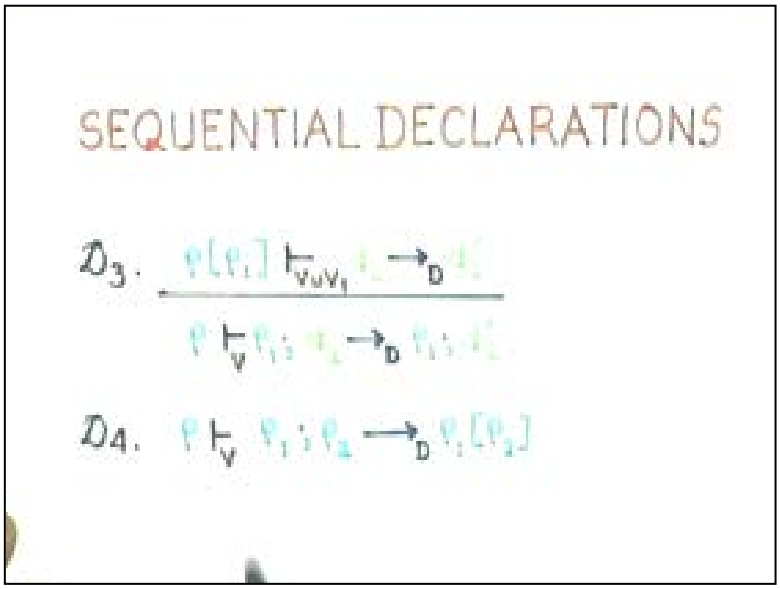
In this case they are evaluated in parallel in the original environment created by the binding and so all occurrences of x on the right hand side actually refer to the x on the other side. In particular you could also have had an expression which contains some x. For example; if you had some x + 1 instead then this reference of x would refer to 3. I could have written it as x + 2 so that it gives the same answer. The reference of x refers to a binding and of course again the binding of ‘let’ extends and therefore the reference to x actually refers to the binding and as a result you get an answer 23. In this case the binding is nested within a declaration and so all references to x in the original expression actually refer to the x in the original environment and since it is a binding nested, all references to x refer to the binding.
Let us look at the semantics of these declarations.

[Refer Slide Time: 06:35]



Let me just briefly look at our notion of semantics of declarations, or definitions as they are called in most functional languages. You have a left to right evaluation mechanism in which based on an existing expression transition system you can define the rules for elaborating elementary declarations. When you have sequential compositions then we look at a left to right evaluation mechanism such that after you have evaluated D 1 you might apply the rule D 2 several times and eventually obtain a little environment for the declaration D1 and having evaluated that then in the updated environment row 1 that you produce, you elaborate D2 and you might apply the rule D 3 several times till you actually obtain another little environment row 2 and then the composition is just the updation of row 1 with row 2.

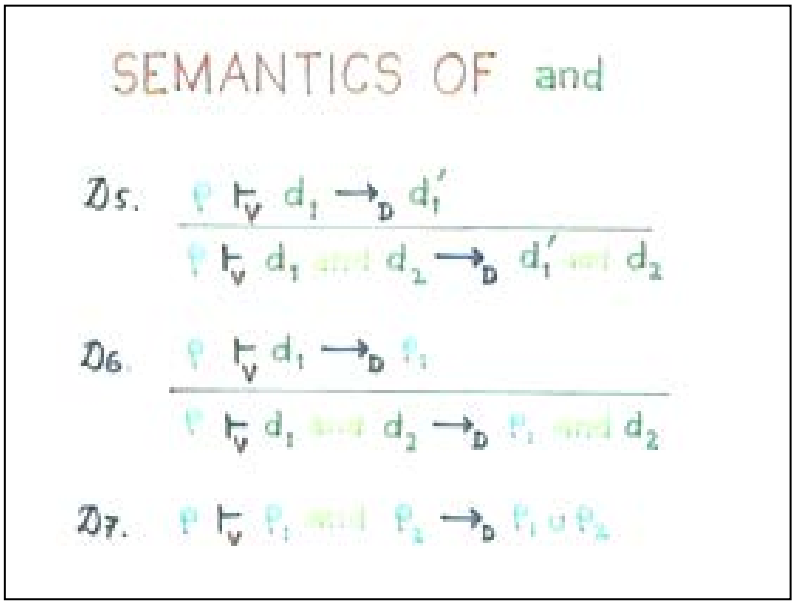
[Refer Slide Time: 07:52]



The interpretation that we gave was that given an environment you have a row.

You elaborate d 1 in row yielding row 1 then you elaborate d2 in row updated with row1 yielding row 2 and the result of elaborating d 1; d 2 in row is row1 updated with row 2. We could similarly give semantics of the simultaneous declarations. Let us look at D 5. Here again given a declaration of the form d 1 and d 2 you have to first elaborate d1 till you have produced a little environment and having produced a little environment you retain that little environment. Eventually when you have produced the little environment row 1 in a standard left to right manner, you actually elaborate the declaration d 2 till you have produced another environment route.

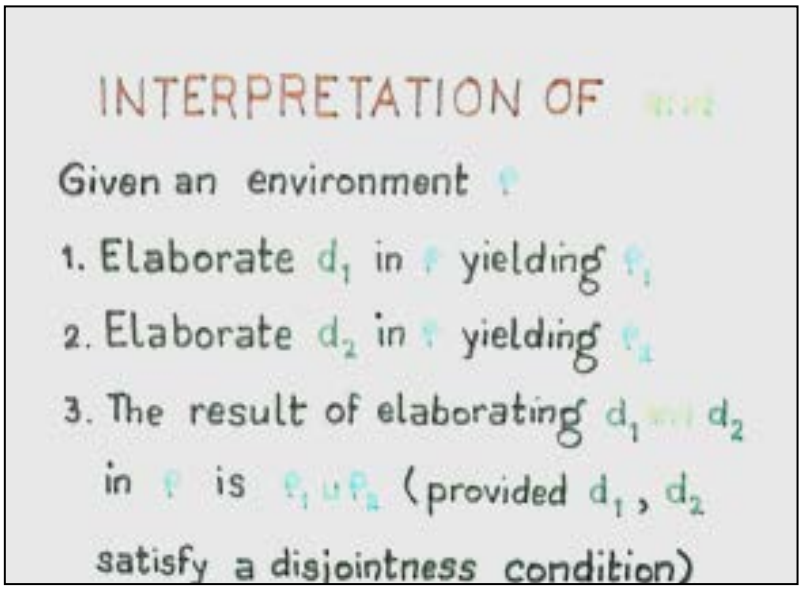
[Refer Slide Time: 09:02]



This elaboration of d 2 also happens only in the environment row and finally it gives you ‘row1 u row 2’ and if the disjointness condition holds, then there should be no conflict in producing the union. By the way the union is pure set union. It is not disjoint. However, if the disjointness condition holds then it does not matter. Disjoint unions, unions and updations all give the same results. We might interpret the semantics of ‘and’ as, given an environment row elaborate d 1 in row yielding row 1 then d 2 in row yielding row 2 and the result of elaborating d 1 and d 2 is just row 1 union row 2 provided d 1 and d 2 satisfy a disjointness condition.

One change that we could make to this semantics is that we could actually look upon simultaneous declarations and simultaneous compositions as evaluating in parallel. However, we have given a sequential interpretation of that where we have taken a left to right elaboration which is not strictly necessary. For example; if you had a very low level parallelism available to you then there is absolutely no reason why d 1 and d 2 cannot be simultaneously processed. Since they satisfy the disjointness condition there is absolutely no reason why they cannot be done simultaneously.

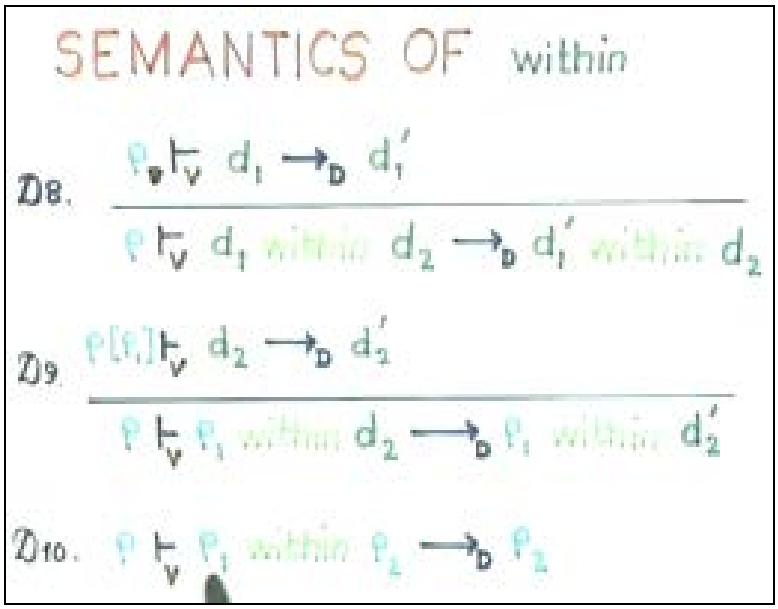
[Refer Slide Time: 10:10]



You could give parallel evaluation rules and in fact the idea of an 'and' means that you could have parallel evaluation. You could have simultaneous movements of both d 1 and d 2 in the environment row. You fork your declarations into two portions, simultaneously evaluate d 1 and d 2, and produce an environment row 1 and row 2 and then join them both by taking a union of the two environments.

Let us look at the semantics of the 'within' declaration. Again we will follow a left to right evaluation mechanism. You can elaborate the declaration d 1 and by applying the rule D 8 several times you would eventually produce some little environment row 1. Then the meaning of d 1 within d 2 is obtained by elaborating d 2 in the updated environment row updated with row 1 and by several applications of the rule eventually you would produce an environment row 2. Row 1 within row 2 just yields row 2.

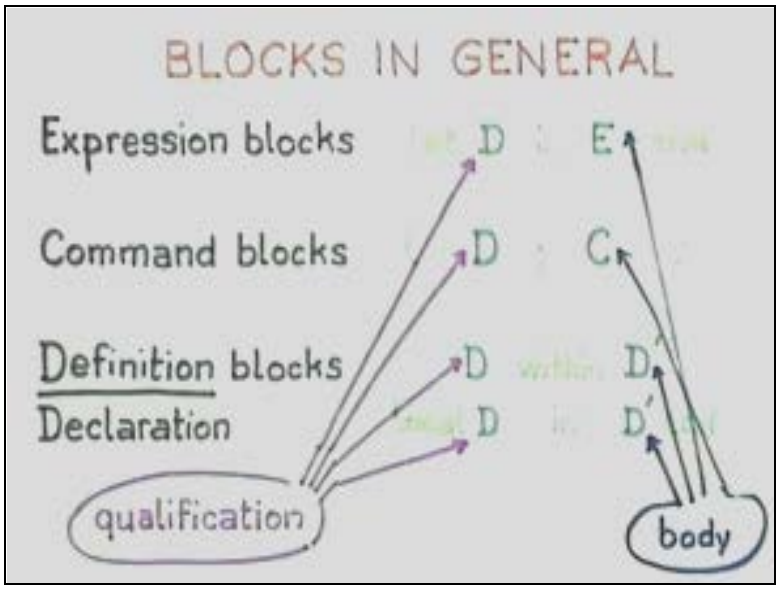
[Refer Slide Time: 12:44]



The effect of row 1 is purely temporary and it acts as a qualification for evaluating the declaration d 2 to produce the environment row 2. So, at the end of the declaration what you have is only a little environment row 2 and row 1 has been washed away.
Given an environment row you elaborate d 1 in row yielding row 1; you elaborate d 2 in row updated with row 1 yielding row 2 and then the result of elaborating d 1 within d 2 is simply row 2.

We have looked at essentially various forms of blocks. So in M L for example the ‘let’ expression defines a declaration to be used in an expression. This expression is actually qualified by a declaration. Then we also looked at command blocks essentially in the same spirit although their syntax was different. You have a command which is actually qualified by a declaration that precedes it and then we also have the ‘within’ construct, which actually in M L, has the syntax local D in D’ end. Our within construct is actually a declaration block or a definition block.

[Refer Slide Time: 15:14]

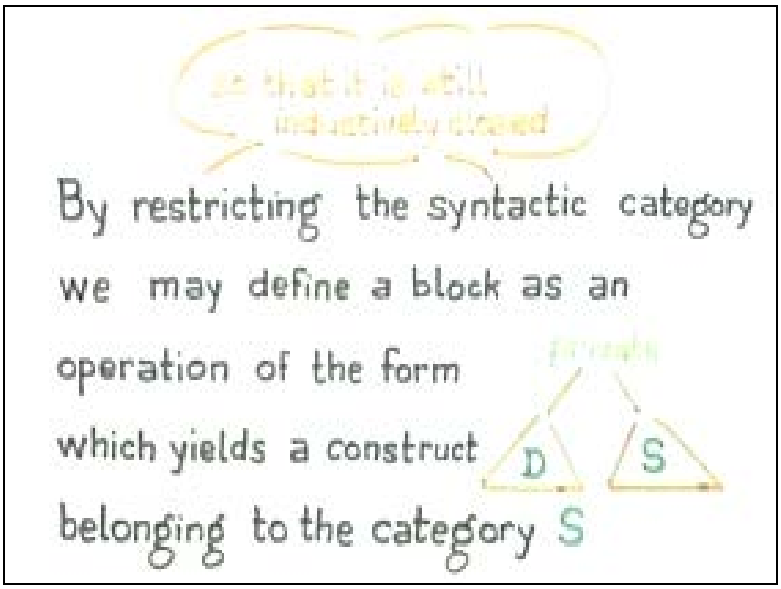


We can talk about blocks in general as some syntactic category with a qualification. So, a block in general has a qualification and a body and the body could belong to any meaningful syntactic category which could be qualified and the syntactic category includes declarations themselves. The meaningful syntactic categories that we are talking about are mostly expressions, commands and declarations. You could qualify any meaningful syntactic category with a declaration and the basic idea is that you have this qualification which is really private to this body, meaning it has no existence outside the core.

It is a qualification that is purely private for the body of that particular block. Regardless of what kind of blocks, we might look upon the block concerned as consisting of some syntactic category. You could even restrict your syntactic category. You cannot put in arbitrary restrictions because it will still have to be inductively closed if you want to write a perfectly general implementation.

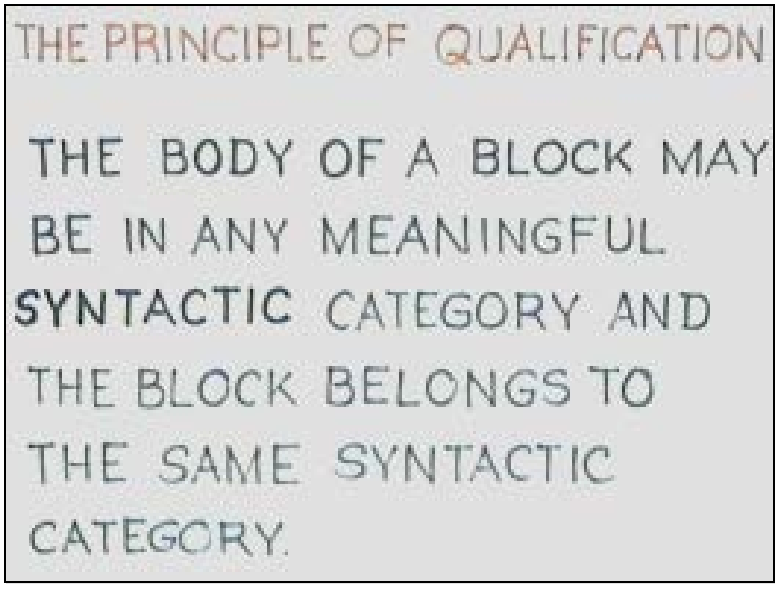
By making restrictions which are very case based what you will be creating in the eventual compiler or implementation is a whole lot of particular cases which really have no relevance. But if our restrictions are such that whatever syntactic category we have chosen is still inductively closed then it means that we can still generate an infinite set of blocks from a finite set of rules and then we might define a block really as an operation in which there is some declaration which is private to some syntactic category. I am using this new word ‘private’ but that is because the syntax of all the blocks is different. I am using a new meta-symbol. There is an operation which says that there is a declaration which is private to the syntactic category.

[Refer Slide Time: 18:18]



When you have this operation, the result of this private operation is to produce another sentence belonging to the same syntactic category. The principle of qualification that is enunciated in tenants book on the principles of programming languages says that the body of a block may be any meaningful syntactic category and the block belongs to the same syntactic category that the body defines. The body of the block really defines a syntactic category to which the block belongs. We will see that there are other ways of changing it and with some other qualifications we might actually change the syntactic category to which a certain block belongs.

[Refer Slide Time: 18:44]

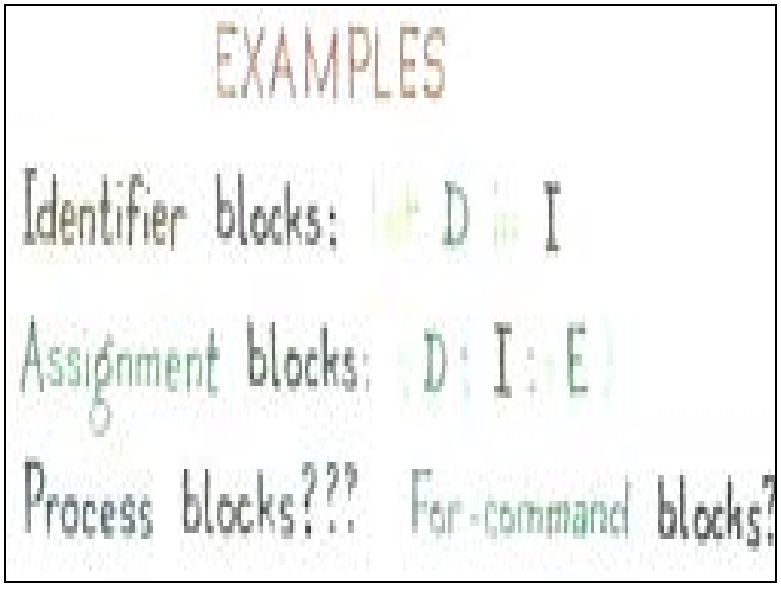


But for the present we might say that a block just consists of a declaration or a qualification that is private to a certain syntactic category and the result of that operation is the same syntactic category to which the body belongs and the principle is important because it means that all kinds of implementation that you might define

are common regardless of your syntactic category. Assuming that you have an implementation of a syntactic category, what happens to a syntactic category that is qualified?

The implementations are all going to be the same essentially. Since we have studied only three syntactic categories you could for example, restrict the syntactic categories in various ways. One is that you could just have identifier blocks; you could have an inductively closed class and identifier blocks. This is a very trivial example but a more important example is that just like you could have identifier blocks and since an identifier belongs to the syntactic category of expressions we would say that it is also one whole expression. You could have just purely assignment statement blocks or assignment command blocks in which you have a purely local declaration. These are possibilities which do not actually exist in most programming languages.

[Refer Slide Time: 22:18]

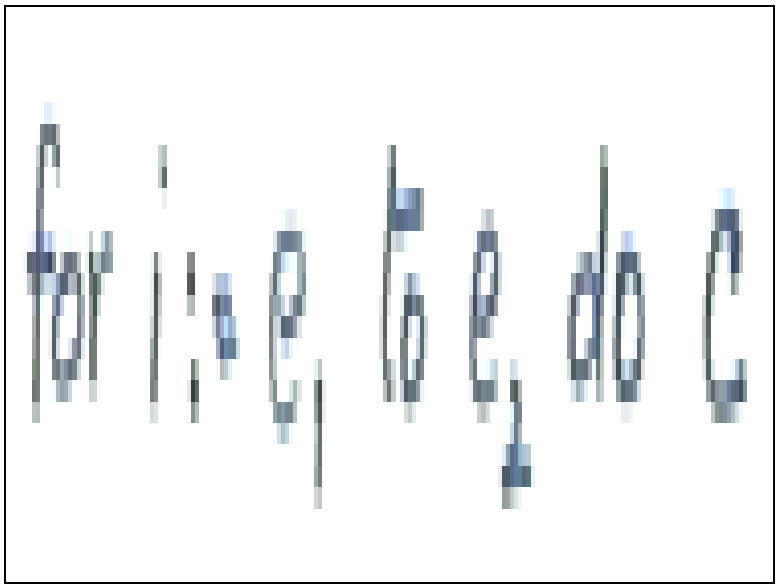


The whole point is that there is no reason why they should not. There are no good reasons why nobody thought of including them. They could just as well have existed but the most important reasons could just be that you do not want to have very small categories because then you will have to do a large amount of runtime memory allocation and de-allocation because of very small scopes involved in such syntactic categories. But there is one important class of constructs for example in Pascal.

If you had defined a syntactic category called process for parallel executions you could actually have process blocks; you could have declarations qualifying our process and you could have process blocks, an important syntactic category. But what for example the Pascal language could have done, which it did not do, was to give a definition for command blocks. In the Pascal language definition regarding the ‘for’ statement, it says that you have a ‘for’ statement which is some expression ‘ for i : = e 1to e2 do C ’. The way the Pascal language is defined is firstly that this identifier ‘i’ is treated as a variable which means that it has to be declared early on along with the declarations in that block.

Secondly, it says that it has to be of a type that is an order type which could be enumerated. It goes from e 1 to e 2; it says that for example, this identifier ‘i’ cannot be assigned inside the command c and it also says that ‘i’ is undefined once you exited the ‘for’ statement. I know that most implementations do not actually enforce this restriction but if you look at the Pascal language reference manual it actually says that the value of this identifier ‘i’ or the loop control variable is undefined at the end of this for statement. Moreover you could use the same identifier i for various ‘for’ statements within the same block. It would have been most natural if this ‘i’ remains undefined before the ‘for’ statement, at least for the purposes of the command ‘c’ where it is undefined before the ‘for’ statement and it is undefined immediately after the ‘for’ statement.

[Refer Slide Time: 24:44]



The idea of having such an identifier as a loop control variable is that it is only going to be a mere loop control variable and is not going to be used for any other purpose. So that is why if it is going to be a pure loop control variable you cannot reassign ‘i’ inside the body and if it is going to be a mere loop control variable at the exit of the ‘for’ statement you cannot use ‘i’ again for any purpose because it is supposed to be undefined.

Then it would have been most natural not to declare ‘i’ as a variable but use a declaration of i as a constant. You can have ‘i’ defined with a sequential composition of a declaration. You could have i defined as whatever is the value of e 1. You could elaborate the declaration i = e 1 and you could use ‘i’ for assignments of other variables but you cannot assign to i within c. This ‘i’ is actually purely local to the block and later you could actually sequentially compose the first execution with the next execution for i : = e1 to e2 do c .

Firstly if ‘i’ is a counting variable then you have to remember to declare it. It might be a very large block. One of the most frequent compile time errors we get is that when we used ‘for’ statement in some large block then you compile it and it says undeclared identifier. That would have got eliminated and you would have clearly

defined a purely local scope. You could look upon this for i : = e1 to e2 do c as a new qualification or as a command qualified by a new declaration that allows you a range of values which are successively obtained. It could have been done in Pascal which was not done because they were obsessed with the idea of a variable and changing its value but we know that constants can also change values.

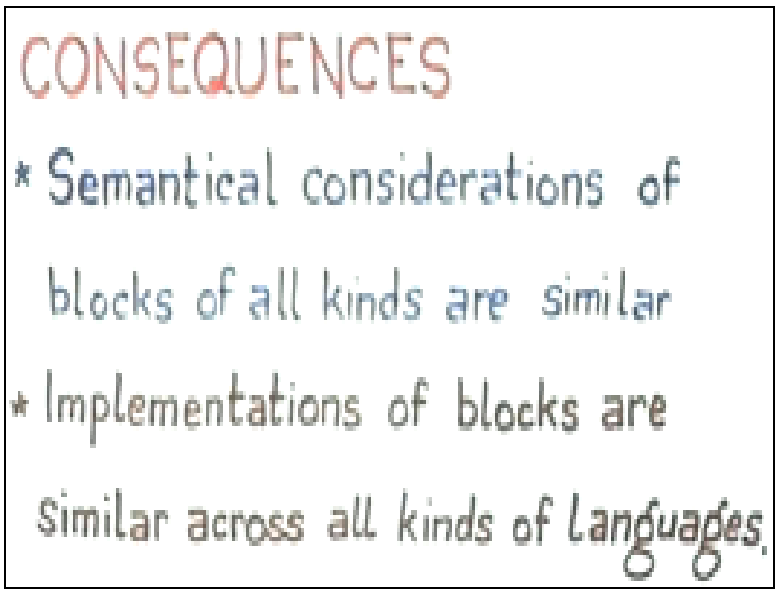
By a sequential composition you get a straight updation of environments and so you could have changed the value of the constant 'i' with every iteration. It would have been a re-declaration in each case but nevertheless it would have made a more natural implementation in which you actually ensure that the identifier 'i' at the end of the loop is undefined and secondly if 'i' was declared as a constant within the loop then there is no way of modifying 'i' inside c whereas in most implementations of Pascal you can actually find holes in their implementation. In many implementations you can actually print out the value of the loop control variable at the end of the execution of a 'for' statement. In many implementations you can actually modify because they do not do any compile time or runtime checks to ensure that i is not modified inside the variable inside the body of the 'for' command.

You could look upon the whole expression as a declaration qualifying a command and therefore defining a new block. Then you would not have to have the overhead of declaring 'i' initially in the block. The 'i' would be locally declared and you could have several 'for' statements with as many variables as possible and there is no cause for confusion because each of them will define its own scope. It is also a way of ensuring that if there is a 'for' statement inside 'c' then the other restriction in the Pascal language is that a loop control variable cannot be used as a loop control variable of a 'for' statement inside the command c. If you treat it as a block it does not really matter even if you had another 'for' statement inside c with 'i' as the loop control variable because that 'i' would be a new 'i' in a new environment. We would be declaring a new constant 'i' in a new environment and within the scope of the declaration 'i' that 'i' would be valid. The moment you exit that 'for' statement 'i' would be valid and there would be absolutely no confusion.

A semantic definition of constructs in a language and of the whole language can actually help you do a redesign or a redefinition of the language. The ones which implement standard Pascal spend a lot of time doing a type checking to ensure that the loop control variable is not used. If you want the value of 'i' to be available outside then declare a new variable that will store the value of 'i' which is declared outside the scope and therefore will be available outside the 'for' statement. The designer of Pascal did not consider the compiling aspect of the 'for' statement seriously.

By all intents if you look at the way blocks are defined it should perhaps have been treated that way. But it of course adds an extra overhead in the sense that you will have to do a reallocation each time for a new block. That is an extra overhead and it is a runtime overhead which means it can slow down an execution. But it would have made the semantics cleaner and it would have made it obvious that if the 'for' statement is a block then obviously anything that is declared inside the 'for' statement, which in this case is just going to be the loop control variable is going to be undefined outside. The principle of qualification has two consequences. Firstly, the semantical considerations of blocks of all kinds are similar and the implementations are also going to be similar.

[Refer Slide Time: 32:28]



Let us look at the pragmatic issues that govern blocks. Let us look at the pragmatics of blocks. We will look upon blocks as a general qualification mechanism and the notion of an environment. I will talk about most of the pragmatics in the context of the functional language because then I do not have to worry too much about the store. Considering the pragmatics of blocks, if we have a declaration in some expression E this expression is evaluated in some environment row and the effect of this declaration is to create a new updated environment and that updation as a reversible process goes away at the end of execution.

Firstly, you have some code that is generated for the interpretation of an expression. This is the code segment and you might also have depending upon what kind of underlying virtual machine architecture you have, a stack as in the case of the PL 0 interpreter. A PL0 interpreter defines a stack in which it interprets each of the codes depending on the values that are at the top. So, there is a value stack and if you have a binary operation to be executed, it takes the top two elements at the stack, calculates their result and places it back on the top of the stack.

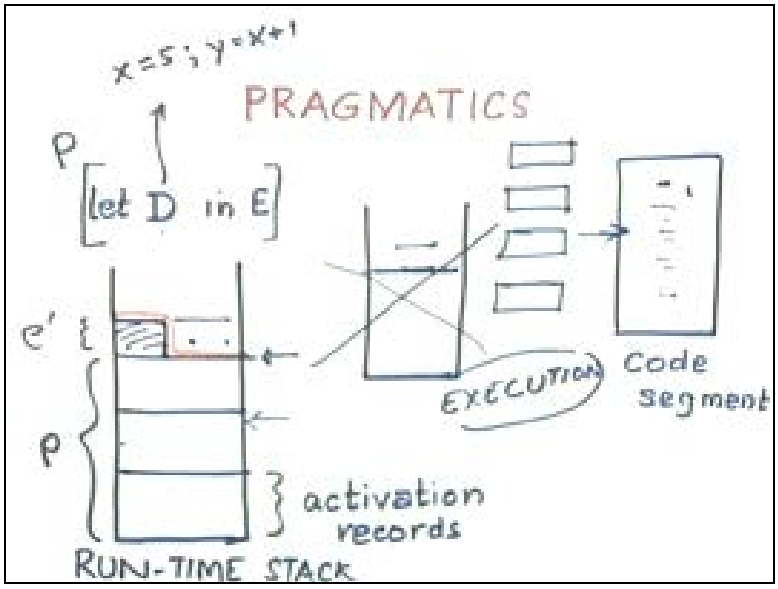
It pops up two values, does the binary operation and places the result back on the stack and thereby executes a code segment. The code segment also could contain load and store instructions. You require an allocation of memory for all the identifiers. This allocation is also done on a stack and a stack is most natural for an environment. You have a reversible process and you want to revert to an original base address of the stack and that is very easy to do. As you define more and more scopes you would have more and more activation records.

The execution stack need not necessarily be a stack. I just mentioned it because in the PL 0 compiler it actually runs on a stack. The underlying architecture is supposed to be a virtual stack-based machine. There is absolutely no reason why you cannot get rid of it and work in terms of a collection of registers and an accumulator which contains the most recent value.

The stack architecture is not absolutely mandatory. You could have a register-based execution mechanism also, in which case the interpretation would be such that you would load them into registers and store the result of an operation on registers into a predefined accumulator. One of the registers would be an accumulator. If you were to generate code for the native architecture for most machines this is what you would actually do.

If your machine is register-based then you would have a collection of registers in which in one of them you would have a program counter stored, in one of them you would have the base address of your runtime stack stored and then you would use some two or three registers depending on all the possible operations you have and one designated accumulator in which the intermediate values are stored as and when the computations are performed.

[Refer Slide Time: 38:33]



Let us consider elaborating a new environment. Starting from a given base address let us assume that this is part of some larger context in which the stack has so many activation records. This whole thing constitutes the environment row. Elaborate this declaration D. Let us assume D has just some two declarations of this form x = 5; y = x + 1. Then as you elaborate D you would first be elaborating x = 5 in this environment. Starting from this base address which is probably either stored in some register or in another stack because you might have to go back to base addresses. You might have several base addresses to go back to so you could have a stack of base addresses also.

Starting from the base address you would allocate some memory and the address is what would be mentioned in your code. So, when you are loading a value onto your runtime stack or into a register the identifier x would have actually come from here. Next when you have y = x + 1 you would first allocate for 'y' a place then in this environment consisting of this much, you would actually execute the expression x + 1. Then you would find a value probably stored in a register or on top

of the execution stack and store that value in this location. So, all references to y in the new updated environment would refer to the address. The address in the stack is what your code segment will contain. Your code segment will usually contain relocatable addresses.

All occurrences of x would actually be replaced by a reference to the current base of run time stack '+ 1' let us say. Since we are considering units as all of the same kind and we have not gotten into complicated data types, it will refer to the current base address '+ 1' which is going to be the meaning of the variable x and the meaning of the variable y is going to be the current base address '+ 2' and the entire code segment in the process of your interpretation of the high level program will just consist of those references. So, the x, the y etc all have been completely eliminated and how do you do that? You create a symbol table in which that address is stored during compile time and that symbol table of course is no longer available.

In your code generation every time you encounter the reference to an x, you look up the symbol table for the most recent definition of x and if it is a variable in an imperative language it would refer to some location and that implementation more or less holds also in a functional language. You would have to still do some memory allocation. After all where are you going to store your constants? As and when new constants are declared you would have to store them. So, pragmatically speaking you would still be giving them addresses in terms of the current base address. During the process of compiling you create a symbol table in which when you elaborate declarations you give an address relative to a base address of runtime stack and that base address depends on the execution environment but you just define a relative address corresponding to a current base address and replace all occurrences of the reference to that identifier by that relative address and when your code is actually generated that is what it will contain.

At execution time the current base address of the runtime stack is known which in the case of the P L 0 compiler, I think, was called b. The current base address is known or stored somewhere and a memory allocation takes place in which the reference is given that value. Then all references refer to the value. There are of course certain speed ups possible.

If you are actually generating code in a native piece of hardware and you have a large number of registers then at compile time itself you might decide not to locate that variable on a runtime stack. If that variable is very frequently used you might actually decide to locate it in a high speed register or in cash. If it is going to be frequently referenced you might actually give it an address in cash. But those are optimizations that need not deter us. The basic principle is that you can look upon parts of the cash and parts of those registers also as part of the runtime stack. It is just that the runtime stack is not one contiguous entity. Then it is distributed across three different entities but it is still a runtime stack.

You might make these optimizations either assigning high speed registers depending on how often it is being referenced in the program or store it in cash but basically there is a runtime stack and you are going to give it a relative address corresponding to a current base. So, it is important to understand this basic concept because this is what we are going to use in all other forms of program declarations whether it is

procedures or functions or co-routines. Whatever you have this is going to be the basic model of execution.

The important points are that there is a portion called the code segment which has no references to any variables. All variable references have been translated at compile time into relative addresses corresponding to a base of a runtime stack. For each block there is an activation record and during the elaboration of a declaration that activation record is actually created. So, the memory allocation is actually done at execution time starting from whatever is the current base. This actually neatly captures the static scoping rules. It does not capture what are known as dynamic scoping rules. It captures static scoping rules. So, the actual memory allocation is done at runtime and all identifiers have been replaced by relative addresses. When you do the actual memory allocation, an identifier instead of having a relative address has an absolute address on the stack.

The moment this elaboration is finished a new activation record has been completed, which is the environment row prime and this expression E is executed in this updated environment row updated with row prime. This whole phase is row updated with row prime and the code for E is then executed based on values taken from this activation record. Either on the execution time stack or on registers the loading and storing happens and the operations are executed. Intermediate results maybe obtained in an accumulator and if there is a storage instruction then that store is to a specific address which has an absolute value and so you can store it back. This is going to the basic model and we will not worry about the execution time environment. In fact we will never talk about it. We will always talk in terms of the code segment and the environment created by an activation record.

In the case of a functional language the difference is just that this whole phase is regarded as just an environment which cannot be updated. Once values are assigned there is no storing back into this run time stack because they are all constants. There is no storing back into this place. There is only the creation of a fresh environment, or when the end of a scope has been reached you just revert to the previous base address and that is it. Nothing that is in the last base address is actually accessible. Everything that occurs is with respect to the current base address and all references are references to some absolute locations in this run time stack. There are no store operations in the runtime stack.

In the case of an imperative language in addition there are stores and updations. Each cell here is actually tagged depending upon whether it is a constant or a variable. So, there is a tag label which comes from the symbol table. The symbol table during the process of compilation not only stores the address but also stores type information and for example we have considered only integers but if you look at other kinds of data types, it also tells you a calculation of how much one unit of that type should be allocated. It also tells you what should be the tag; is it a constant or a variable? All that information is present in the symbol table and a lot of that information is actually present as tags in the activation record.

If it is a constant you can do a runtime check to ensure that you are only updating a location which has not been designated to be a constant.

It does not have a tag of a constant and you can update only locations which have that tag of being variables and you have standard memory allocation strategies within the runtime stack.

We will talk about the heap later but this is the basic model of execution which the environment gives us and the contents of this runtime stack actually give you the store.