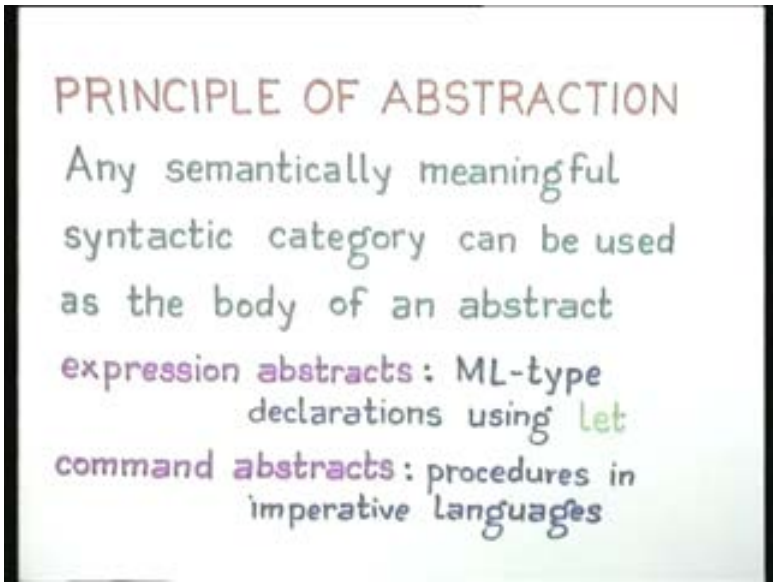
Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 37
Procedures

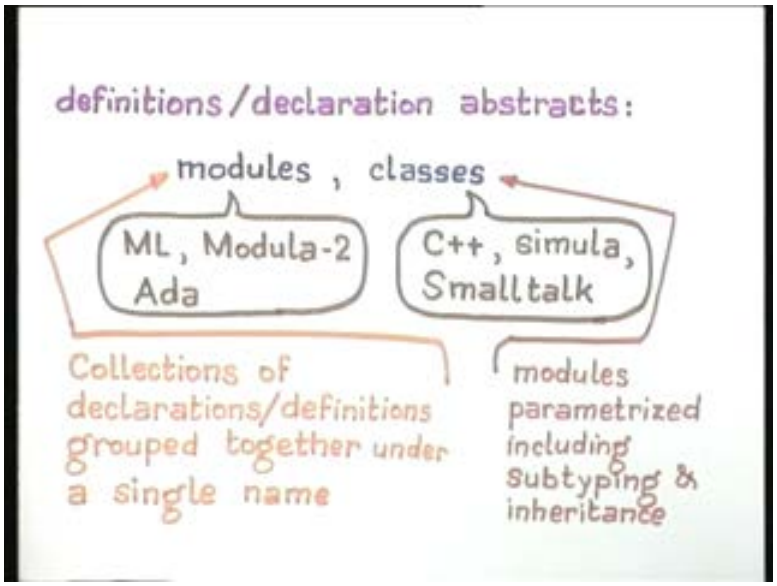
Welcome to lecture 37. We will continue with procedural abstraction.

(Refer Slide Time: 00:36)



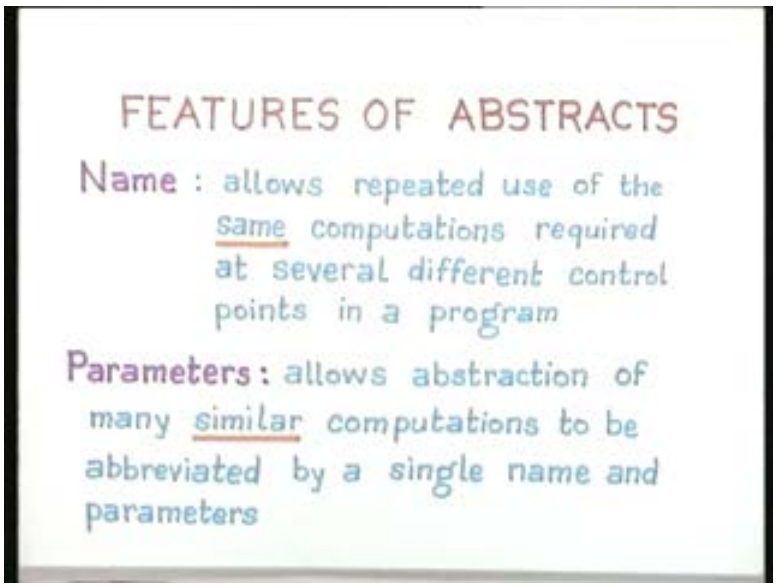
As I said, one thing is that the whole purpose of abstraction is that you can take any semantically meaningful syntactic category and you can use it as a body of an abstract and an important part of using it as a body of an abstract is to give it a name. So essentially when Shakespeare asked what's in a name the answer should have been an abstract but he apparently didn't get that.

(Refer Slide Time: 01:02)



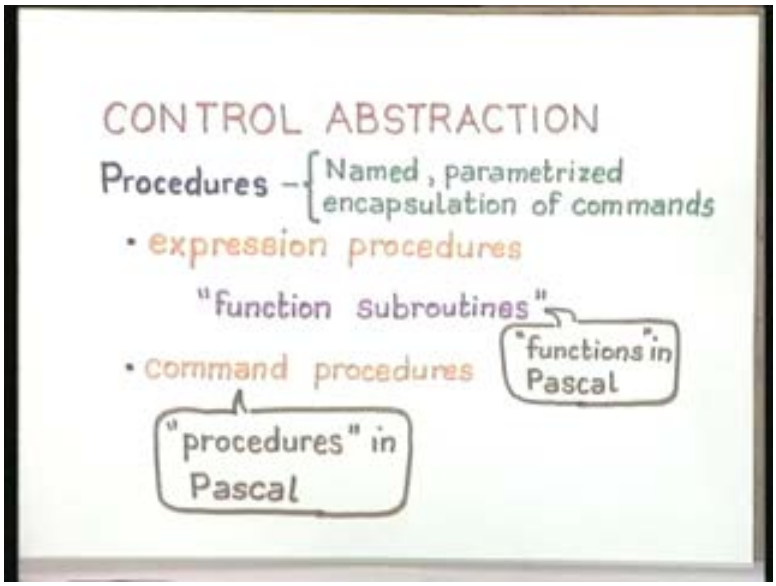
You have various kinds of abstracts. We have partly seen what expression abstracts are, then you have procedural abstracts which are commands, command abstracts and then you have definition or declaration abstracts as in Modula, ML, C ++, Simula, Small talk etc. Therefore the most important features of an abstract are really its name. Of course we have to answer the question what exactly is an abstract, that's an important semantic issue that we have not yet addressed and of course the parameters to capture, similarity of computations and so on and so forth.

(Refer Slide Time: 01:50)



Then you have a type checkable body which is elaborated whenever the name is so it is to be called with appropriate parameters. We were concentrating on control abstraction and what I say for control abstraction namely procedures more or less also holds for these expression abstracts where the expression abstracts denotes functions.

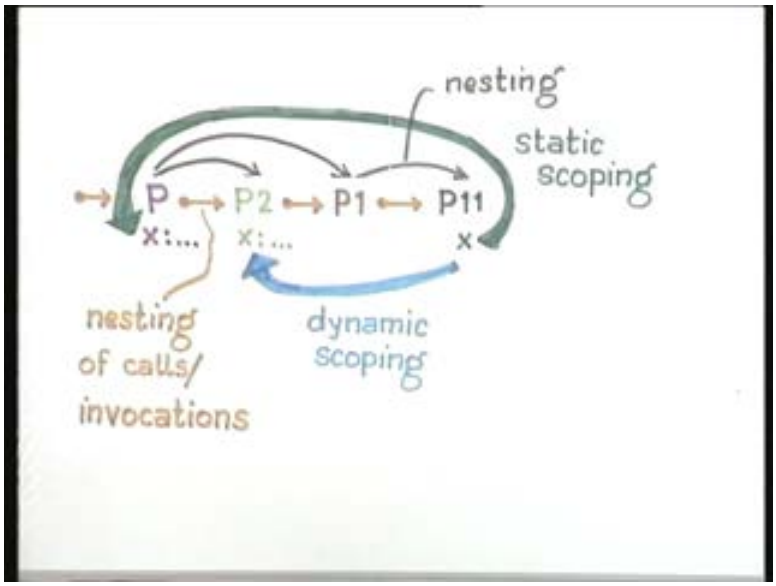
(Refer Slide Time: 02:28)



Now let us look at the form of control abstraction that we have got which is basically like functions in Pascal or procedures which change state in some fashion. Of course when it comes to these kinds of abstracts normally there are two kinds of scope rules that you might use. one that we have been using mostly in languages like Scheme, ML and Pascal, C, Aida, Algol sixty most of the languages are compiled rather than interpreted though of course the ML environment is an interpretive environment but most languages which are compiled prefer to use a static scoping mechanism which means that it is easier to debug and look at non-local references from the text of the program rather than from the run time environment. So the bindings are all compile time or the identifiers are all statically bound and d in the case of dynamic scoping what it means is that non-local references as in this case where you have a chain of calls assuming this large program P has two procedures P1 and P2 which are at the same level of nesting and independent of each other and within P2 there is a call to this independent procedure P1 and within P1 there is a call to its own nested procedure P11 which has a non-local reference.

So if it were a statically scoped language this non-local reference would refer to a binding occurrence that occurs in the innermost textually enclosing block. On the other hand in a dynamically scoped language this reference would actually depend on this calling sequence and would refer to the most recent invocation of a block which actually contains a binding occurrence. So that would be the difference between static and dynamically scoped languages. And the reason I said that dynamic scoping is mostly used in an interpreter style rather than in a compiled style is basically because of the run time environment.

(Refer Slide Time: 05:16)



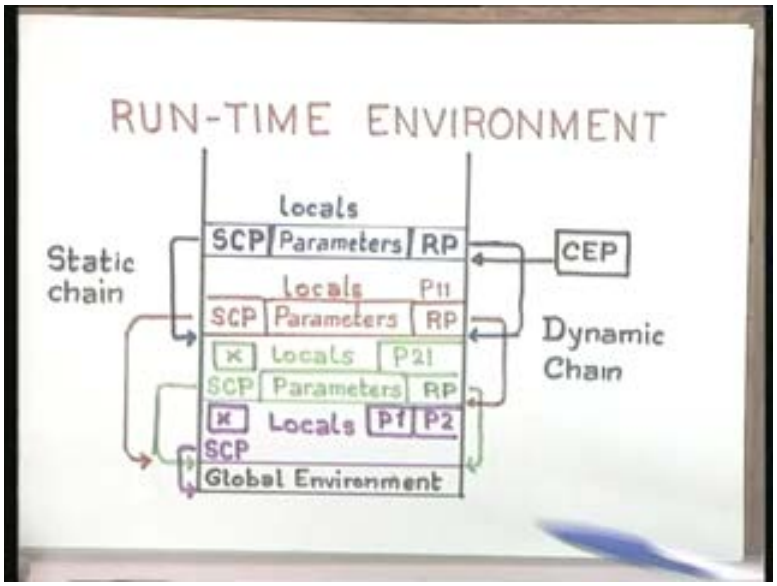
Let us look at this program executed at some control point here after this chain of calls. So if you look at the run time stack of this program the activation records actually look like this. Firstly of course you have a global environment this is sort of normal since no program usually works without a global environment you usually require some global environment which contains standard IO procedures may be libraries and so on and so forth so let us assume that there is a global environment in which every program executes. and of course then there is this call to procedure P1 there is this program P which has a pink border or a purple border if you like so the activation record for this program P has a local declaration of a variable x and it has the procedures P1 and P2 declared locally.

Therefore in the process of compiling what would actually happen is this P1 and P2 would refer to the jump addresses to the code segments of the appropriate procedures. so the fact that P1 and P2 are local to the program P means that you have to somehow create a jump address the address of the code segment and then P1 and P2 could have parameters so in any call within P to P1 and P2 you will have to type check parameters. So a part of this would also be some information on kinds of parameters in the order in which they appear and basically their types so that any call to P1 or P2 when you are actually compiling this main program should type check. In the run time environment this P1 and P2 would refer mainly to the code segments of P1 and P2 where the jump addresses are or where the codes start from.

Hence from within P of course there is a call to P2 which is this green block and so there is an activation record for P2, P21 is of course nested within P2 so there has to be some address to its code segment and there is a local variable x and there are of course parameters but this is the main program so it doesn’t have any parameters.

Strictly speaking main programs also have parameters which usually refer to something in the library routines like a Pascal program actually refers to the files that it is going to access but let us assume that there are no such things. So a procedure usually has some parameters, it has local variables may be locally defined procedures the address of whose code segment you require to maintain and then from P2 there is a call to P2 so there is an activation record for P1 which is similar and from P1 there is a call to P11 so there has to be an activation record for P11. So the typical run time environment when you reach this control point is that there are four activation records calling chains of length four besides the global environment.

(Refer Slide Time: 09:26)



Therefore now what happens is this. Every time the moment you stop executing P11, supposing the execution of P11 is completed then what happens is you are back in the environment of P1 which means, here this is the current environment pointer so you have to know at each stage what is the new environment pointer that you have to jump to when you exit this scope during execution. So when the scope P11 is exited during execution then this top blue block should go and your current environment pointer should point to the bottom of this.

So the fact that you are creating a new little environment at each stage means that you have to have what is known as a return point which will return you to the new current environment pointer each time you exit a block at execution time. During execution the moment the block P11 is exited you require a return pointer which gives you the base address of the previous activation record. Thus this RP is called the return pointer and whether your language is a dynamically scoped language or a statically scoped language you require these return pointers each time. So what is called the dynamic chain which is really the calling sequence?
If you remember the calling sequence, so this is really the dynamic chain.

So there is a calling sequence so you have a dynamic chain which points from P11 to P1, P1 to P2, P2 to P and P to the global environment if necessary which is not shown here. So essentially you require this dynamic chain which is maintained at execution time.

Now, in a language that is interpreted it is easier to implement a dynamic scoping mechanism, for the present let us assume that this is not there. In a language that is dynamically scoped what it means is that every time you see a full phrase you translate it and execute it before you look at the next syntactic phrase. Now what happens is supposing you see this phrase containing this non-local reference this blue reference to x then when you have to translate it you have to check out whether you can just traverse down the dynamic chain to find out the first reference where x was actually declared or where x has a binding occurrence.

Since translation and execution both take place simultaneously at the same time essentially most of the basic information about identifiers and their bindings and so on will still have to be maintained in a symbol table at run time. Since there is no compile time each phrase is textually read translated code is created and executed. So whenever you read a new non-local reference you don't know anything about it till you traverse down the dynamic chain. Either you have it in the current environment or you have it in the previous environment or you have it in the previous environment whichever comes first you just bind it to that. so under the dynamic scoping rule or dynamic bindings that means all the translation is done at execution time there is no separate compilation phase then what happens is with just this dynamic chain the innermost enclosing invocation which contains a declaration of that non-local reference can be found by just traversing down this chain and searching through the subsequent activation records.

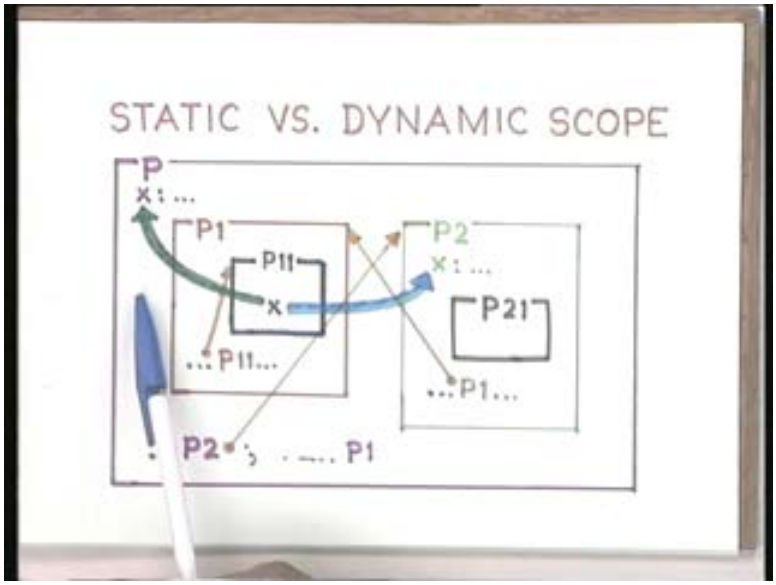
Hence, if you have a non-local reference what makes it a non-local reference is just the fact that there is absolutely nothing corresponding to this activation record which contains a declaration of x or a specification of its type. So what do you do is you go through the previous invocation which is P1. You check whether in the activation record of P1 is there some identifier with a declaration for x. If there is not then you just traverse down from the return pointer of P1 to the previous thing and check and you will find may be there is an x declared here. Remember that all the symbol table information, identifier, name, its type and everything since it is interpreted has to be stored somewhere at run time. Thus an identifier with a certain name is possible at run time provided you do everything at run time.

In a compiled language identifiers and names no longer exist after compilation and they all have been translated into addresses so there is no question of string matching identifiers and so on and so forth. In a dynamically scoped environment you just keep traversing down this dynamic chain in order to find the most recent binding occurrence that was encountered in the chain of calls. So this dynamic chain serves the entire purpose which is the reason early implementations of LISP and APL wanted to make it interpretive, it is also an easier thing to write or actually use this dynamic scoping mechanism. But what it means textually when debugging a program is that you cannot

look at an abstract you cannot look at a procedural abstract in isolation and hope to get anything out of its non-local references.

The only way you can understand abstract is by essentially hand executing and producing the appropriate calling chain by hand execution and then determining what would have been that reference at that point what would have been the binding occurrence for that non-local reference at any point. Interactive debugging may be easy but a planned debugging of the text is usually quite hard with a dynamic binding mechanism. So most languages like ML and Scheme which allow for compilation is because they actually would like to encourage debugging from a text they implement what is known as a static binding mechanism which means essentially that kind of binding mechanism that one sees in Pascal. But then what does this mean?
This means that in the run time environment. What happens in a compile language?
This is specific to compile languages.

(Refer Slide Time: 19:26)



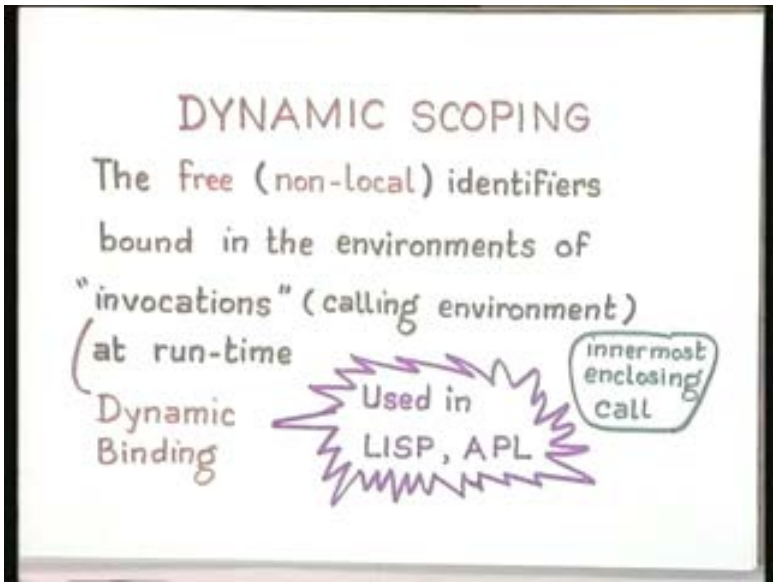
At compile time a symbol table is created and an address a relative address relative to some base for each identifier is created. What is the translation of an identifier?
An identifier denotes a simple variable, it is just the replacement of that name by its appropriate relative address with a reference to the scope in which it was bound. So, in the process of compilation I can actually create an address for this with the depth of nesting 1 and a relative address which basically says that this has been allocated so much of memory so many bytes from the top of the base from where this scope would start in activation record in the run time.

Now any non-local reference x here would therefore refer to just this address with the appropriate nesting depth with the same relative address which is actually a reference to this x.

Hence, as you read through as the compiler actually scans the token file and it encounters this identifier x it just hashes on to some binding occurrence and translates this x by the appropriate address. So the code that is actually executed only has relative addresses which at execution time the bases will be filled up. As and when activation records are created there will be a base address stored somewhere.

As you have seen in your PL0 compiler there is a base and the stack for the current activation record and all addresses are relative to how much above that base should you go before you reach this variable. So what this also means is that since you have addresses which are related to a base in the case of the run time environment you have to maintain the static structure of the program somehow you have to maintain for example the fact that both P1 and P2 are nested within P so P1 is this red activation record it has a pointer to the pink one and P2 is this green one which has a pointer to this pink one and this blue P11 is nested within P1 so there is a pointer from the blue to the base of the red so that now (Refer Slide Time: 09:26) if you can maintain this information then all non-local references which have a nesting depth comma relative address given to them means that you go down this appropriate chain rather than this chain you have a current nesting depth of 2 and you have a non-local reference x which you have a current nesting depth of 3 and P11 is at nesting depth 3 and you have a non-local reference which has been translated as having nesting depth of 1 and has so and so address relative to that base of that nesting depth 1. This means that you have to traverse down two steps and go up to the appropriate translation above the relative address in order to access x. So this chain of textual nestings is what has to be maintained in the run time if you have to have a statically scoped language implemented especially a compiled language. So a typical language with dynamic binding will just have a dynamic chain if it implements static binding then it also has to have a static chain point which essentially captures the nesting levels and the textual containments of the various blocks.

(Refer Slide Time: 23:54)



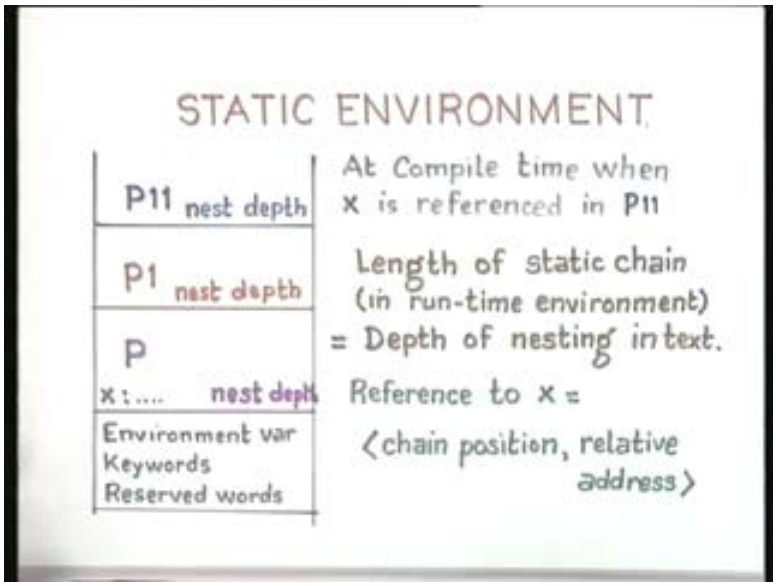
So, essentially the difference is that in a dynamically scoped language. The free or the non-local identifiers are bound in the environments of invocations calling environments at run time. Dynamic binding is much more easily implemented in interpreters you just have to follow the return pointer chains.

(Refer Slide Time: 24:25) Here this SCP stands for Static Chain Pointer, this RP stands for Return Pointer once this invocation is over what is going to be the current environment pointer or what is going to be the new current activation record and what is going to be its base address so that you get from here. So essentially when you exit this block this return pointer the value in this location is going to be copied into the current environment pointer, the value in this is just the address to this base and so your current environment pointer will be pointing to this. And all addresses that you will be referring to will refer to addresses which are anyway less than this. So you don't need to explicitly garbage collect it with further invocations. All the previous activation records which are really not supposed to be active and that have been left behind as dead wood will get overwritten.

The P1 and P2 are the pointers to the P1 and P2 in the main program and the reason why we are elaborating P1 and P2 is that, under either a static or dynamic scoping mechanism when you actually read P1 you will be actually creating its code segment somewhere and it is going to be a pointer to that code segment. Mostly it is going to be a pointer to that code segment. Very often if they are procedures with parameters what happens is that for type checking purposes you don't want to put all the information about parameters.

Basically what you want is type information and the order of the parameters. You may not want them all to be all located here and since different procedures have different numbers of parameters with different kinds of types what you might do is you might standardize it to two pointers. One is address to a code segment and another is a pointer to some place where the sequence of parameters and their types and whatever information you might want to include about their types is maintained so that you can do the type checking. So it is some record which basically allows you to find the code segment of that procedure if you want.

(Refer Slide Time: 27:19)



So what it means is that under a static binding mechanism for a compiled language as you compile as you read the token string after lexical analysis basically it's a token string for each block that you have created you actually maintain a nesting depth the addresses for each of its variables, variables are going to be translated by the addresses so the symbol table contains type information, address, size this that and such things.

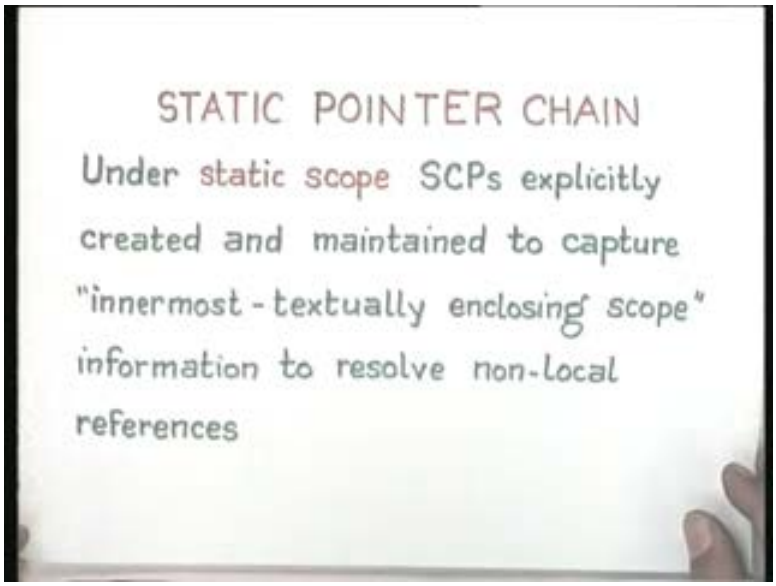
Therefore at compile time you have this symbol tables for each of these and as you textually read the program you actually create all this information that is essential. After compiling P11 the moment you have exit P11 this part of the symbol table is no longer necessary and you work with this part (Refer Slide Time: 28:22) especially when you are reading the body of P1. And once you have finished reading the body of P1 then this part also goes away and the next textually available thing is the procedure P2 so a new symbol table for P2 is created like green in color and then within P2 there is a procedure P21 maybe black in color and as you read P21 you create another symbol table for P21, the moment you exit P21 the symbol table for P21 is removed and you are left with symbol table for P2, the moment you have finished generating the code for P2 which essentially means you have finished reading it and generating code P2 goes away and then you are back with just this which is the main body of P of the program and you can elaborate the program.

The symbol table here should also contain some information saying that there are these two procedures P1 and P2 because in the main body of P you can have calls to P1 and P2 which have to be somehow type checked. Similarly in P1 you should have some reference to the procedure P11 because you can have cause to P11 and you would require a type checking.

The way you would create addresses is that at every point in the symbol table you will actually maintain what is the current nesting depth that you are looking at and for each

identifier essentially the current nesting depth is going to be equal to the length of the static chain which is going to be maintained at run time. So any reference to a non-local variable or even to a local variable will be the chain position and a relative address corresponding to the base of that activation record.

(Refer Slide Time: 30:56)



So these static pointer chains will have to be explicitly created and maintained in order to capture this innermost textually enclosing block information to resolve both local and non-local references it is a uniform procedure.

Now let us get back to this diagram, if you look at the Static Chain Pointers if you look at this there are actually two different Static Chain Pointers which of course meet at the main program, they are two completely independent. One is this green single pointer chain and the other is this double pointer chain. So at any point during the execution of the program where I am assuming a statically scoped language the only references you can have to identifiers are identifiers in the current activation record, identifiers in the activation record immediately pointed by the Static Chain Pointer or identifiers of the activation record pointed down by the Static Chain Pointer of this Static Chain Pointer. Therefore what I mean is the only references you can have in this blue procedure are two identifiers in the blue procedure, in the red procedure or in the pink procedure and you cannot have any references to identifiers in the green procedure under a static binding mechanism.

Now you can think of it if you just generalize this diagram you have lots and lots of procedures P1 P2 P3 P4 P5 all at the same level and all of them having very deep levels of nesting with more and more procedures embedded in them then your run time stack when it is executing some piece of code in the innermost nesting of some procedure can actually have a whole lot of Static Chain Pointers which are all independent except for the fact that they meet at the main program.

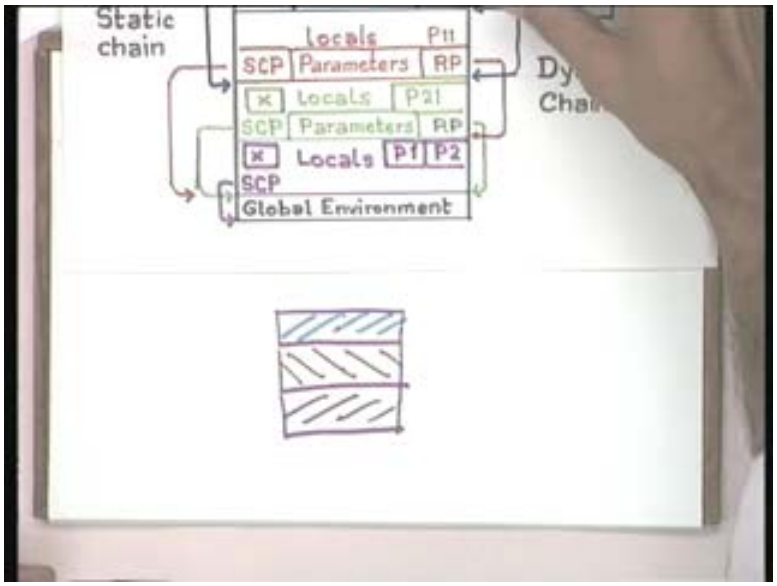
You actually have a collection of disjoint chains they are not disjoint because they all meet at the main program but otherwise they are all disjoint. So at any point you might have a huge collection of static chains of which at any instant only one static chain is really necessary for your access in order to access non-local references. You could have a huge number of independent static chains but at any instant there is only one static chain that is actually useful to you for executing that block.

Here the example is that there are two independent static chains and at this point this static chain is the only thing that is necessary. If you consider a calling mechanism for this same program such that you call P2 and from P2 there is a call to P21 and from P21 of course there could be a call to P1 and there could be a call to P11 then you would have two independent static chains and depending on which block you are executing you require only one of the static chains to be available for that block, that's the only static chain that is useful for that block and the moment you exit that block you might move into a different block which is not textually nested which means you will require a different static chain.

And in languages where there is a huge amount of side effects like most imperative languages what it means is that instead of having a huge number of parameters for their procedures lot of people tend to use a large number of globals. In that case for a really complicated program what it means is that traversing down the static chain for each and every non-local reference can become expensive and can slow down execution.

So at any point what happens is very often some speed up mechanisms are used and one speed up mechanism is the use of a display. so a display is just a collection of high speed registers some how organized logically in the form of a stack which we use in some order and since out of a vast collection of disjoint static chains at any point you require only one complete static chain pointing down up to the main program. These Static Chain Pointer addresses are all stored in the display in a last in first out fashion. So the base address of the pink block will be stored here and then the base address of the green is not important, the base address of the red one would be stored here and the base address of the blue one would be stored here.

(Refer Slide Time: 38:19)



As long you are executing this blue block these are the only base addresses you require in order resolving non-local references pretty fast. you know the nesting level, textual nesting level here is 3 if you find a non-local reference at nesting level 1 or some such thing then all you have to do is look at the base address given here to down this display stack and look at the relative address starting from this base address.

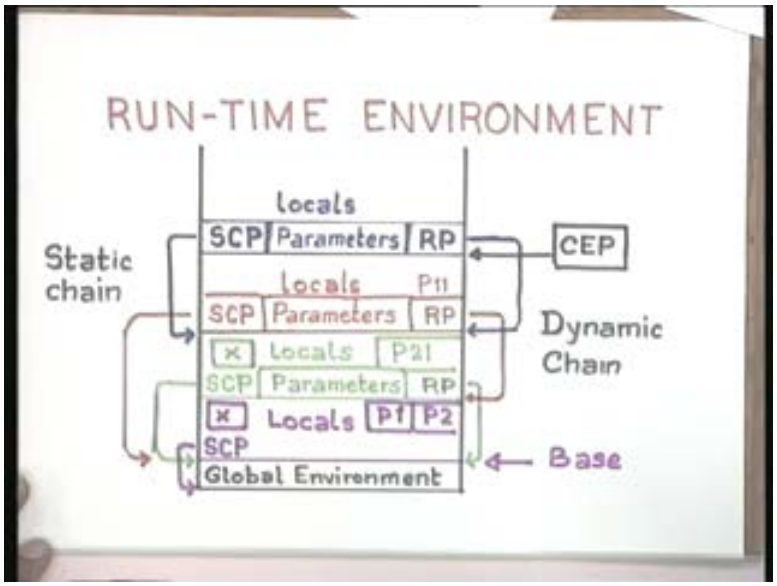
Just imagine if the stack actually grows and you have a reference to a global variable at nesting level 20 that means you will be doing twenty hops down the static chain before you can access that non-local variable and you don't want to do the twenty hops, that entire procedure or function might have no parameters nothing and it might just be doing some task which is completely modifying global variables which means for each global variable occurrence that is there either for modifying it or for reading its value for each of those global variables you will have to make twenty hops down this static chain and then find the address and either modify it or read the value, you don't want to do that it can slow down your execution tremendously. Thus by the use of this display what you can actually do is you can just subtract nesting levels. From the current nesting level go to the display to find the base address and find the relative address from that base. So essentially what we are saying is that this pink actually contains the address of this base address.

(Refer Slide Time: 40:23) This pink base the address of this base is actually given here similarly the address of the red base is given here, the address of the blue base is given here and these are the high speed registers which make accesses fast. Anyway that's a sort of an optimization or a speed up if you like it is nothing more than that. So the act of naming essentially does this. What it means is that in the case of a dynamically scoped language there are enough overheads just translating the phrases at run time and accessing them. But you don't have this extra overhead of a static chain. You just keep following the return pointers which form the dynamic chain every time you want to get to

resolve a non-local reference. So there is an extra overhead with static binding but then that extra overhead means that debugging your program is easy, it is possible to compile programs use them for production runs whereas an interpreted programming language basically means that you will be doing on the fly not really debugging you will be doing only on the fly, redefining, on the fly development which may not be systematic which could be quite adhoc so the extra overhead is worth it provided we can some how speed up global accesses. And speeding up global accesses means using some method by which pointer hopping can be saved somehow using a fast access.

Supposing you have got a nesting level of 60, 20 or something you may not have actually that many registers but let us assume that you have cache which is a fairly high speed memory mechanism then you can access the cache. But the point is that you are at a nesting level of 16 you got a global variable at nesting level of 1 and you want to find out its address you don't have to do sixteen hops down the static chain but you just have to go to the cache fifteen places down random access because this is going to be organized as an array if you are going to organize it in cache.

(Refer Slide Time: 45:24)



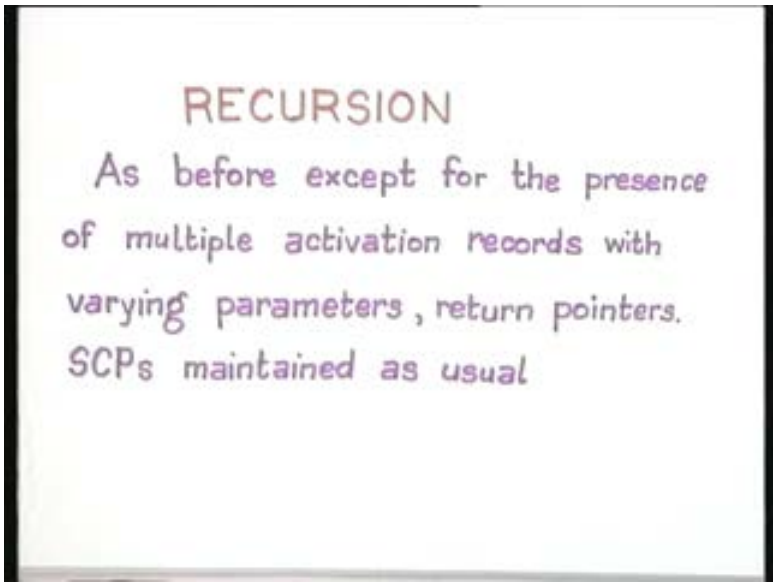
These are all just addresses so they are all uniform so you can actually randomly access the corresponding base address go back to the run time stack to that base address and take the relative address from that so your access speeds are substantially improved when you use a display mechanism and hopefully your execution will improve.

What is the address of that x?
That x at compile time the x has been translated into something that gives the nesting level and relative address from what ever is corresponding base. That's the job you are doing at compiling. While compiling what are you doing about non-local references?
You are replacing all identifiers by a nesting level cum relative address with respect to the base of that block. So your main problem at run time is finding the base of the block,

your relative address is there and how do you find that they are the base of the block? It so happens that the difference of the nesting levels is just going to be the length of the Static Chain Pointer the number of hops down the static chain that you have to perform. So, instead of doing that number of hops you actually organize that static chain itself as an array in some high speed memory so that you can randomly access that. Therefore each time when your static chain changes there is this extra overhead that the new static chain will have to be copied out into this display.

For example, here the moment I exit this blue block on exiting this blue block I am not going back to an activation record which is along the same static chain path. I am going into an activation record which belongs to a different independent static chain. So the extra overhead when I exit this blue block can come to this red block is that I have to completely erase out my display and now before executing this red block before continuing with my execution of this red block what I have to do is I have to copy out this static chain into the display to enable quick accesses to non-local references in the code segment of this red block.

(Refer Slide Time: 47:40)

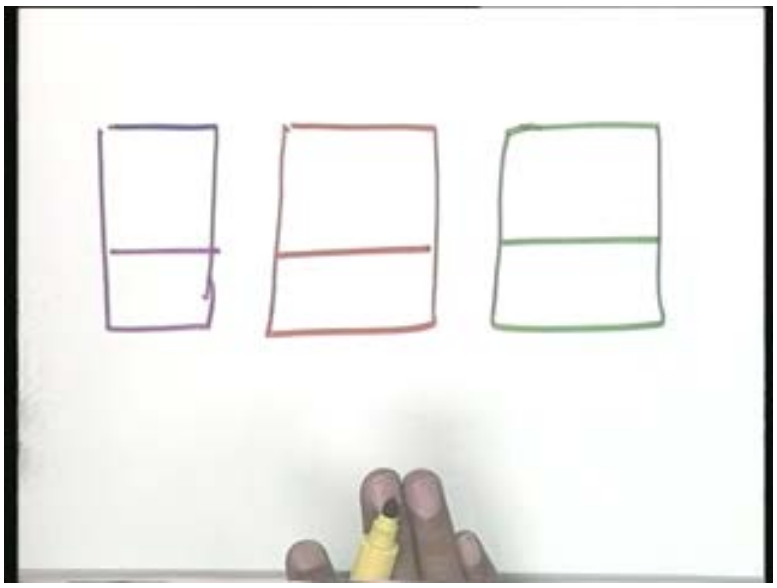


In the worst case it could narrow down to actually traversing the thing. For example, when you go from here to here it just works out to that but that's not something that can be really be guaranteed. You have to at least do one traversal down the appropriate static chain fully before you can decide what to do and it is easy to do that blindly by just copying out the display every time you exit the block and enter a new block. So when I exit this blue block and come here I look at the Static Chain Pointer copy it out into a display in a reverse fashion and then traverse this, copy out its Static Chain Pointer and so on. But every time I exit a block and get back into some previous invocation I have to do a complete traversal of the static chain at least once. But what I am claiming is that is all you have to do then you can save on subsequent hops through a static chain by using a display.

We are now looking at pragmatics before semantics because lot of the pragmatics is very easy the act of naming has not significantly changed implementations. And in the case of recursion it is even trivial. One important thing about naming I said was to provide a form of abstraction and important feature of that kind of naming as a form of abstraction is also that it allows you to call recursively automatically. For example in an unnamed block except with explicit “go to” statement you cannot go back to the beginning of the block from within the block but with naming you can automatically make recursive calls. And the whole point about implementations are once you have organized your run time stack in this fashion you should throw your mind back to the run time environment in languages like Pascal and ML as opposed to run time environment in languages like Fortran.

If you recall what we did in Fortran, in Fortran every block was independent of each other and they were all statically allocated in memory and they all contained persistent data. And any reference had an absolute static address. For example, if you were to take a diagram in a Fortran like environment with static allocation what it means is that there is going to be a pink block code segment cum data separate from red, P1 is going to be code segment and data, P2 the question of nesting of course never arose so let us assume P11 and P21 do not exist, P2 would have code segment and data and they would all be fixed for life that is for the lifetime of the program.

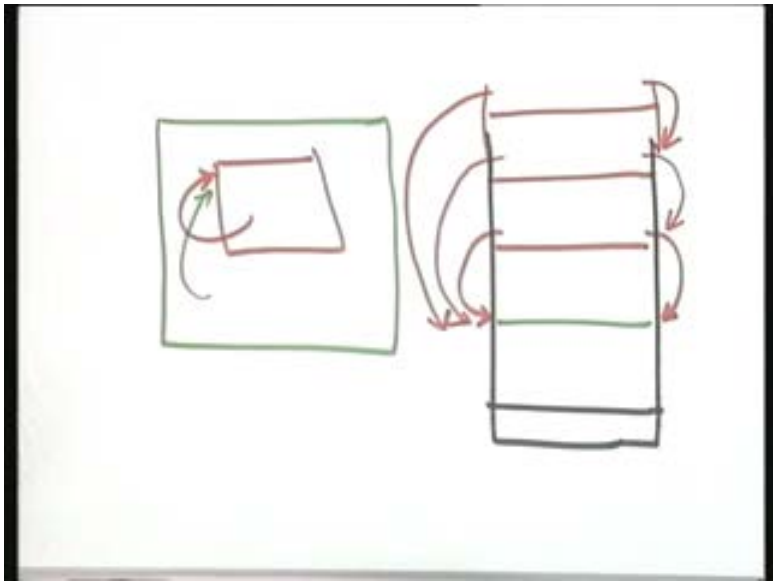
(Refer Slide Time: 50:04)



So what it meant was that all references whether local or non-local would have a fixed absolute address which is relative to the address given at loading time of the program relative to some base given at the loading time of the program. So what that means is that you cannot have recursive calls because recursive calls means having different copies of the same activation record with different values and so on and so forth. But the flip side of the coin is that it is also very fast having fixed at all Fortran programs execute

extremely fast because they don't have these complications of allocating fresh at run time, de-allocating, traversing, pointers none of these so the executions are very fast. This is one reason why scientific programs are still written in Fortran and the other important reason of course is that lots of them were written in Fortran and people are feeling too lazy to change it. But the moment you organize your run time environment as in the form of stack with a dynamic allocation of memory of activation records and so on and so forth you directly open up and make it flexible for the purpose of implementing recursion.

(Refer Slide Time: 53:34)



So what it means now is if there is a recursive call you have this program here then there is another sub program here which is recursive then what it means is that your activation record would really look like something like this. For over n calls to this subroutine what you will have is essentially one activation record for the main program without a return pointer, Static Chain Pointers and then you have several activations of this recursive call with a Static Chain Pointer pointing here and a return pointer pointing here. And then let us say that you are assuming the case when the recursion has been called thrice then you will just duplicate these static pointers like this and your return pointers will be like this.

You can have several incarnations of the variables local references will be locally resolved non-local references will go down the static change. When you exit one invocation you automatically have from the return pointer the address of where the invocation is that you have return to so there is absolutely no problem with the implementation of recursion in this case. The next important question is what exactly we mean by meanings when you have functions.

(Refer Slide Time: 53:49)