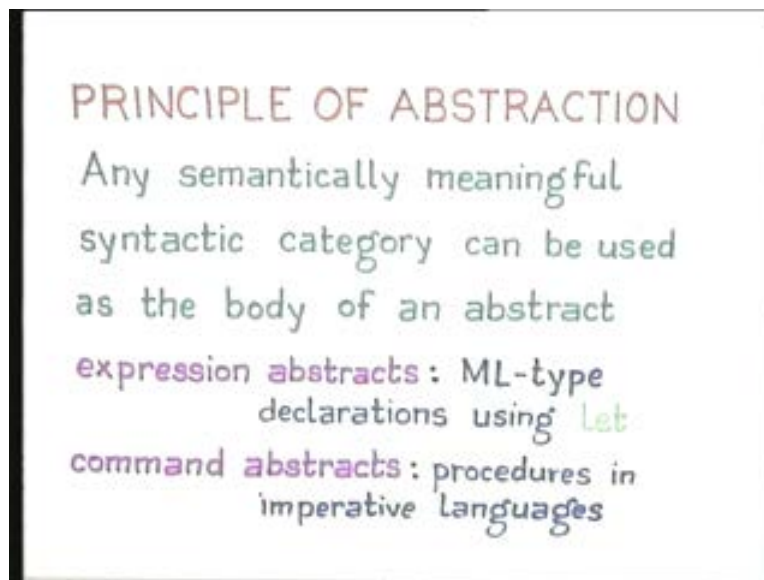


Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 36
Abstracts

Welcome to lecture 36. So today we will start abstracts, it is not a very common name for what might be called an application of the principle of abstraction which again is not a very common principle. Essentially according to Tenent for example in his book he defines the principle of abstraction as “any semantically meaningful syntactic category can be used as the body of an abstract”.

(Refer Slide Time: 1:00)

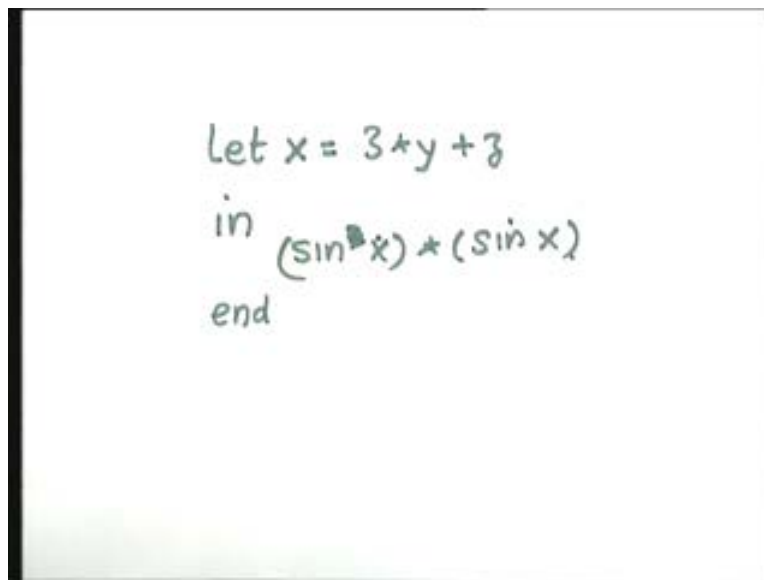


Actually what we mean by an abstract?

We have actually done some abstraction in a certain way and that is the declarations in the ML let construct. So, essentially the whole point about abstraction is this. Let us not confuse this word abstraction with the abstraction in the lambda calculus though they are some what related to. So when we talk about abstraction we are really saying that there is some complicated object or a group of objects and in general I am talking about any object it does not necessarily mean code, it does not mean necessarily data or code. So it is any object in fact which we would like to group together either for some logical reason because they belong together or we would like to group them together for some convenience. Therefore for example a department within an institute is a form of abstraction where there is a group of people, equipment, furniture, rooms whatever which somehow are grouped together and regarded as one consistent whole.

For example, records also form some part of an abstraction in programming languages. But the notion of an abstraction can be generalized to essentially not just data but any logical grouping of objects. I don't necessarily refer to objects in the sense of programming objects; I am saying that we use this principle essentially throughout our lives in every field of activity. Thus one natural abstraction is the declarations in a let construct in ML. The main the main body of a let construct is the expression which is finally evaluated. But there is enough reason to take subexpressions from that expression abstract them out by giving them a new name which is what you do in a declaration and use that name wherever that subexpression is intended. So that is a form of abstraction.

(Refer Slide Time: 6:15)



```
let x = 3 * y + z
in (sin x) * (sin x)
end
```

You have some complicated expression, you split it up into subexpressions which for various reasons either for purely logical reasons or because you want to frequently use that subexpression, for various reasons which might be determined by a programmer he removes that subexpression puts a declaration and gives that subexpression in name and uses that name wherever that subexpression is intended. So essentially what the programmer has done there is that he has performed an abstraction in the sense that as far as the main expression in a let construct is concerned he would like it to somehow stand on its own. So you have some declarations like this; let x is equal to 3 star y plus z in (sin x) into (sin x) which is some complicated expression.

Essentially what has been done is that this expression 3 star y plus z should have occurred here wherever x occurs and for some reason best known to the programmer he has deemed it not exactly necessary but convenient at least to abstract out that subexpression give it a name and use that name throughout. Therefore, essentially what this abstraction gives us is that it gives us the main body of this expression where if we choose we do not really require looking deeply into what exists. It provides a form of hiding which you need to look into only if you really want. So that that's what an abstraction does.

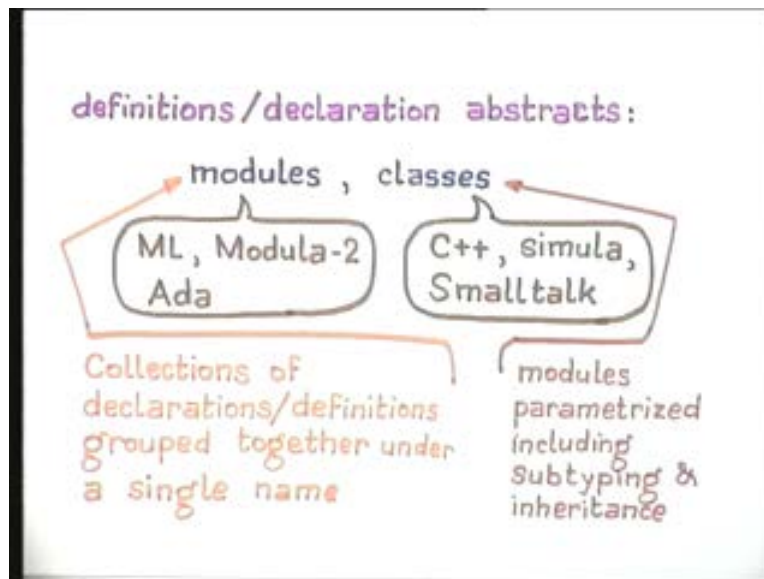
An abstraction takes some complicated object or group of objects and somehow performs a certain hiding on them so that what might be perceived as irrelevant or unnecessary detail can be hidden or can be eliminated or can be skipped and only the main overview of that group of objects can be viewed and this abstraction is something that is actually very common throughout our lives.

If you look at all your pieces of equipment essentially what you are saying is that there is an internal you have got a black box view and the black box view itself is a form of abstraction. You got an interface which is all that is visible to you, the internal details are hidden the internal mechanisms are hidden and all that you are interested in is the interface. So you don't need to open the black box if you don't want to know the internal details. So the internal details are hidden.

A form of hiding that we have already seen is that of scope rules but that's a very rigid form of hiding in the sense that if you are inside the scope you can see whatever is there in the scope but if you are outside the scope you really cannot see anything. Therefore scoping is also a form of abstraction but what we are looking at more now is what might be called named abstractions. We have looked at abstractions in their unnamed forms either as unnamed blocks or lambda expressions but what we are looking for now are named blocks so named abstractions.

And naming is despite what anybody might say is actually very important despite what Shakespeare might say it is actually very important. If the flower rose did not have a name Shakespeare wouldn't have been able to give his famous quote about it and he would have puzzled what to do about it. So naming is actually an important aspect of all forms of abstractions.

(Refer Slide Time: 9:50)



So the major syntactic categories which are for us semantically meaningful throughout that is a unified view of programming languages would say that there are really only three syntactic categories; expressions, commands and declarations and all of them have abstractions available to them. You can create abstractions with all of them and what we will primarily concern ourselves now is with what might be called the command abstracts so that are really procedures in imperative languages.

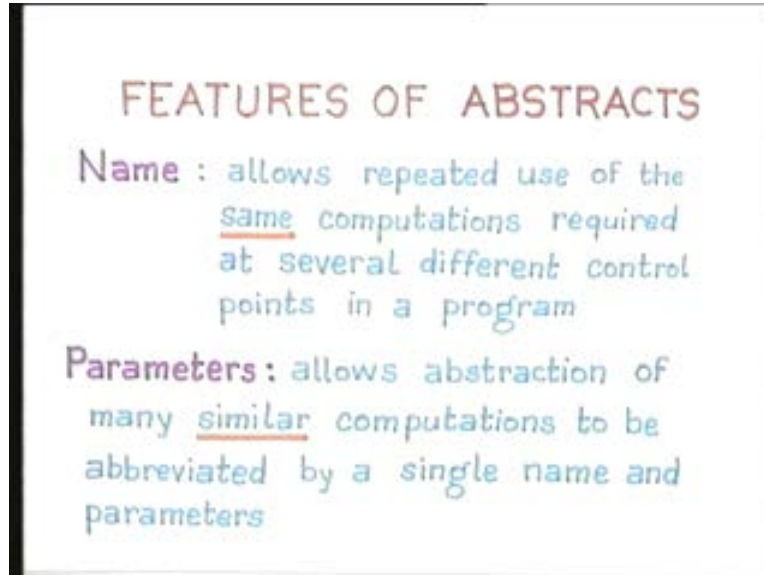
We have already looked at expression abstracts though I did not explicitly mention that they are abstracts. We will now look at procedural abstracts which are not exactly the same as the notion of procedural abstraction which Ableson and Sussmann give in their book on scheme but it is similar and related. So we are looking at abstraction in imperative languages and then there are abstractions possible over declarations or definitions and they are the abstractions which are called modules or classes.

Modules are just a named collection of declarations grouped together for some reason. A class is a module parameterized to include various subtyping and inheritance properties. These are the declaration abstracts.

Now let us look at procedural abstracts. The most important feature of an abstract is naming. So the fact that you give a name to an abstract means firstly that even if that name is inappropriate, incongruous or plain ridiculous it still stands for some collection of properties or objects which it names. And naming therefore allows a repeated use of the same kind of objects in an abbreviated form.

A naming actually provides an abbreviation which can be used repeatedly. So our use of names even in natural language or use of pronouns are really forms of local declarations so that you give something for the name “it” and you can use the word “it” repeatedly for that something and that something could be some complicated object which may not be capable of being described very conveniently and it forms a logical role. Thus the use of words like “he” “ate” “she” and so on in our natural language is really a form of local declaration and very often most people use them ambiguously. But if they are not used ambiguously they actually form a local declaration to allow for repeated use. So pronouns are really names for abstracts even in natural language.

(Refer Slide Time: 16:44)

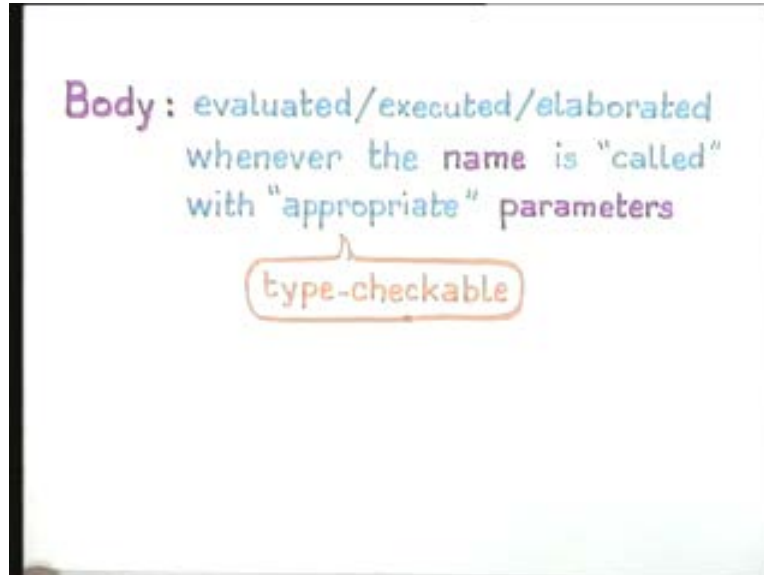


Therefore naming is an important thing in the sense that it allows the repeated use of the same computations in the case of programming languages at different control points in the program. That is in fact true of this expression abstract too. There are at least two occurrences of this x and if this x were something really complicated (Refer Slide Time: 14:49) then I would have to replace this x by that complicated expression. So it is important to have naming so that you allow an abbreviated abstract name which can be used repeatedly in different contexts.

Essentially for the same kind of computations you want to perform at different contexts by the use of a name you can abbreviate that. But semantically speaking the moment you introduce a name it also means you have to introduce a binding. And pragmatically speaking the moment you introduce a name you have to generate some code which actually interprets that name in something that is consistent with the semantical use.

Therefore, both semantically and pragmatically names create fresh bindings and we have to do something about those fresh bindings. But they provide this convenience to the user of repeated use of actually standing for some complicated object or a group of objects and so can be referenced by single name and further if you parameterize the names then instead of just being able to duplicate the same computations by using their name you can duplicate many similar computations by varying the parameters. So a name and a collection of parameters allows you to actually perform this abstraction so that the parameters form what you might call the interface to the black box that is actually externally available for viewing without looking inside the black box.

(Refer Slide Time: 18:53)

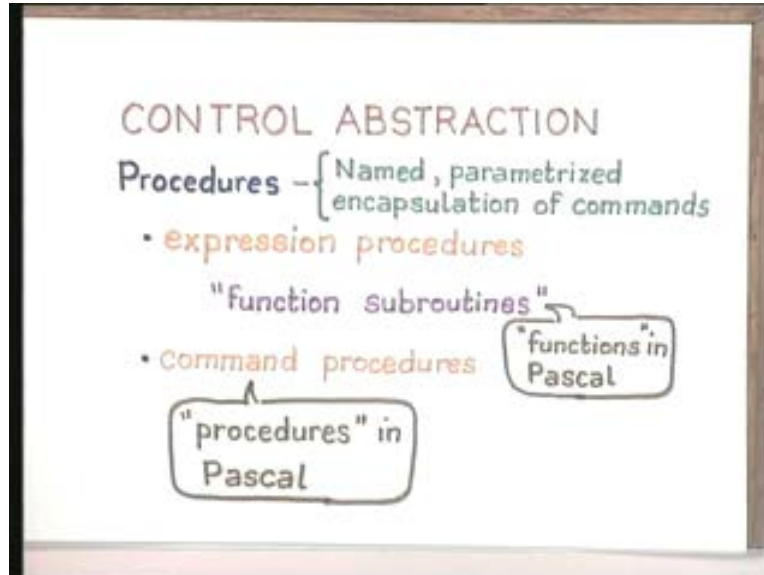


When you look at many similar computations to be parameterized you are performing another form of abstraction where you are emphasizing the similarities of the computations and deemphasizing or hiding the differences in those computations. So, abstraction as a form of hiding essentially with parameters means precisely that. You are trying to emphasize or highlight the similarity of different users of that name rather than emphasize the dissimilarities or the differences between the various users in the various contexts.

Especially the issue of naming has cropped up recently also in the whole question of mobile computations with cellular phones, car phones. The specification of such things turns out that philosophically, linguistically and computationally it is one of the most important objects which we have overlooked or treated quite shabbily in the last forty years of programming languages. So the main features of an abstract are its name, parameters and of course what it actually represents and is its body.

You can have expression abstracts that means the expressions can be evaluated, you can have command abstracts which means those commands can be executed, you can have declaration abstracts which means those declarations can be elaborated. Hence the three features of an abstract are really this; the issue of naming, the issue of parameters and the body. We will not worry really too much about the body because we have already done unnamed blocks and abstractions and we know how those bodies are executed or elaborated or evaluated. Now it is just a question of looking at naming and parameterizing. If you look at parametrizing then the question of parametrizing again boils down to substitutions in a calling environment. It is a matter of performing certain appropriate substitutions in a calling environment. The issue of naming is also a question of performing a substitution in an appropriate environment. Eventually all forms of computation are really forms of appropriate substitutions in appropriate environments.

(Refer Slide Time: 23:05)



So that's one of the reasons why we had to do the lambda calculus in some detail so that you appreciate that everything really has to do with substitution. These are the features of an abstract and we will concentrate mainly on naming and control abstraction for sometime. Let us look at control abstraction.

Most programming languages are what might be called a command abstraction. So a command abstraction in most imperative programming languages is really in the form of procedures. What are these procedures?

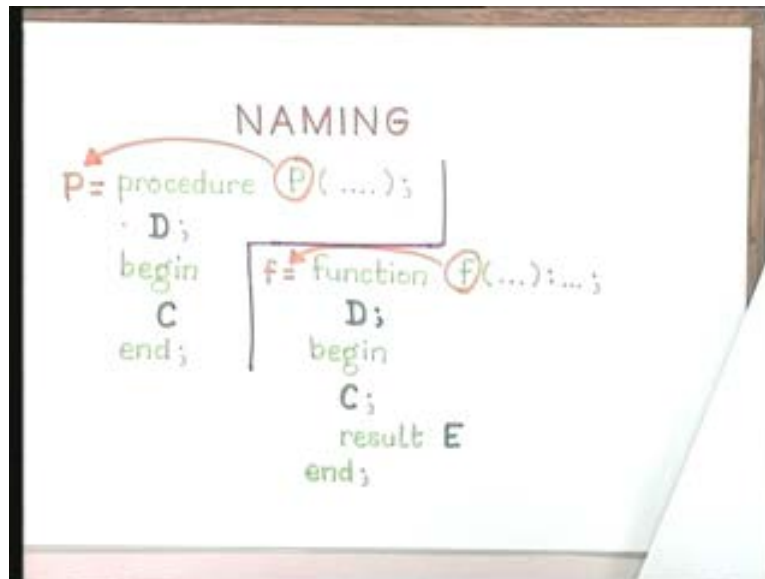
They are really an encapsulation of a command or group of commands and by encapsulation I essentially mean gift wrapping it so that you don't see the internal details, you might name them and parameterize them. Naming of course is compulsory in many languages but parametrization is optional.

In languages like Algol sixty even naming is not compulsory and that's how you get unnamed blocks. In Algol sixty C you can use unnamed blocks so that naming is not compulsory. And essentially what happens in such languages is that naming is used only when there are repeated uses in different contexts. If there is going to be only a single use but it is still an abstract you just define an unnamed block. So encapsulation of commands is really what command abstraction is all about. This is some terminology which I have taken from tenant, it is not necessarily widely used but it actually captures the essential meaning that you have expression procedures which basically corresponds to functions in Pascal.

Functions are a greatly abused word so it is a good idea to use the term expression procedures. They are procedures because they transform state because they are commands their bodies are commands so they are a command abstraction, they are expression procedures because they return values of expressions too. They are procedures because they are a command abstraction and therefore they change state or change

configuration but we prefix them with the word expressions because they also return values. And then you just have what might be called command procedures which are the normal procedures in Pascal.

(Refer Slide Time: 27:00)



Basically entities in Pascal with the reserved word function are expression procedures, entities in Pascal with the reserved word procedure are command procedures. And procedures in general the way we will be using the word is to denote any form of command abstraction which has a name is important because it means it also has bindings. There are bindings to their name and you have to decide what exactly is being bound to that name, what is the type of object that a command procedure is, what kind of a what kind of a creature is it that's an important question and the name should denote that, **should classify the kind of beast that this command abstraction is.** So what we will do is let us play around with a few permutations.

Let us look at an abused Pascal like syntax and I will tell you the reason for the abuse. Let us look at a Pascal like procedure. For the moment I will forget about the parameters they are optional. So a typical procedure structure is reserved word procedure some are named, a procedure identifier, parameters may be semicolon, some local declarations begin a command end.

So now what have we done by this naming is that we are claiming that this is the name of a semantical object which represents this command abstraction. So the syntax for performing the command abstraction is the reserved word procedure. So essentially what we are saying is let `P` be the abstraction procedure parameters, local declarations begin `C` end or the let `P` be the name of this particular abstraction with these parameters.

Similarly, for functions you have the syntax and what we are essentially saying is let `f` be the expression procedure defined by function parameters, colon, some type, semicolon,

local declarations, command, and result E which I have written for the sake of convenience and the reason for that is as follows. The way Pascal works is that it uses the function name itself as a local to the function. So instead of this what you would have is f is assigned E. But then there are two f's in the same scope. One f is this beast which we still don't know what it is, it's an abstract, the other f is a local variable which just has its types specified here. There are other problems with it.

In a language in which you have side effects global variables can be used can be modified within the function so on and so forth and in a language it does not allow more than a simple type to be returned as function values supposing I want a huge collection of values to be returned record to be returned or even a file to be returned after the evaluation of a function. What do I do?

One possibility is to actually return a pointer to that object which is what for example most C programmers do they return pointer to structure which you can do in Pascal too. The other possibility is, if it is some complicated type then treat it as a global and perform the side effects and make it a parameterless procedure. or in case there are problems about whether it was successfully executed or not just return a Boolean value but the actual effects that you are interested or the values that you are interested are stored through side effects on global for example on files.

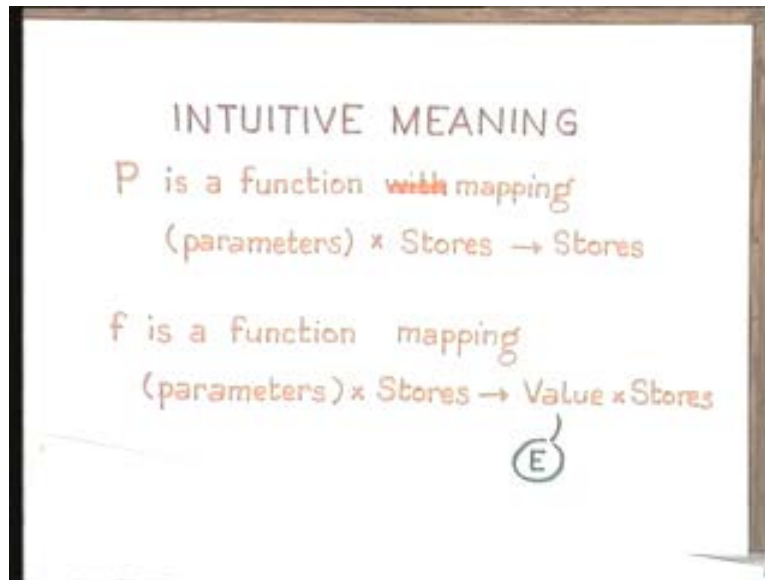
Now, supposing there were no parameters supposing it is a parameterless function and I had f assigned some expression involving f itself then there is a resulting ambiguity that is that f a recursive call to this function f or does it refer to the local variable f which was created as part of the elaboration of this function? so supposing if I followed rigid Pascal syntax I had this statement like this for a function f which is parameterless then the use of the same function name as a local variable I still don't know what the type of an abstract is but what I do know is that the type of this regarded as a local variable is whatever is the type specified here. Since this type is involved inside here it is not at all clear that this function itself has the expression abstract f as the same type as this local variable f and why should it? it may not, in which case does this f refer to the local variable f or does it refer to a recursive call to the function f itself, that is an ambiguity that is not really been addressed. Many implementations take their own view.

For example, turbo Pascal actually refuses to recognize a recursive parameterless function. It assumes automatically that if the name of the function occurs on the right hand side of an assignment then that f must be the local variable f which is created as a part of the function. That is one reason why it is perhaps not a very good idea. Of course there are very simple syntactic ways of rectifying this ambiguity and that is to insist that all functions whether they have parameters or not have a pair of braces around them in their declaration. And if you are using it as a local variable just use f, if you are actually making a recursive call make it this way.

There are simple patches of changing this but that's not the point. The point is that what you are interested in firstly is that this object which is being called f whose type I still don't know all I know that it's a command abstract whose most important things are the

creation of some side effects because it uses commands changing state and finally returning a value of the type specified here and that's really what this object is.

(Refer Slide Time: 34:06)



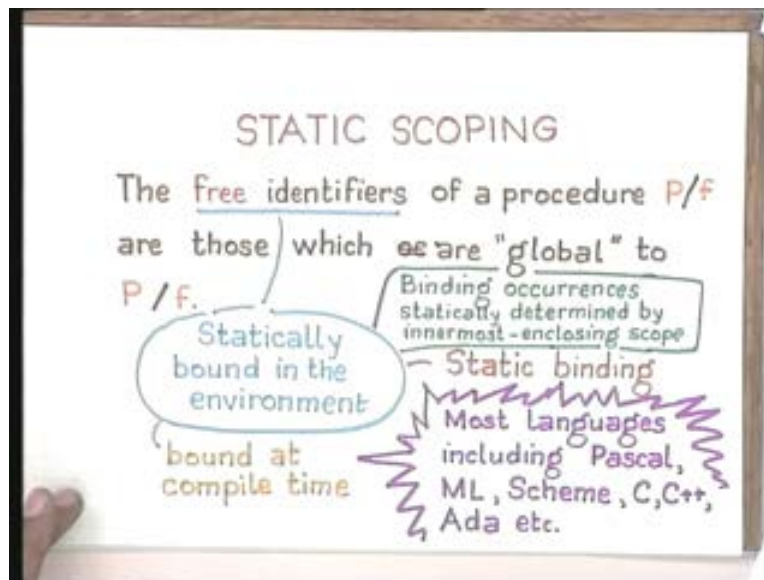
Here this P and f are with reference to the previous example. Essentially the intuitive meaning of a procedure in a imperative language is that a procedure P is a function which might take some parameters and a store and gives me a fresh store. And an expression procedure or a Pascal function is a mapping which might take some parameters and a store give me a value which is actually specified by the body E of the expression procedure and give me a change store which is actually defined by the command C . and since the language allows for side effects even in expression evaluation this E itself could change the store the evaluation of this E itself could change the store.

Therefore, an expression procedure or Pascal like function is just something that takes parameters of appropriate types and the current store and returns a value and a new store. A procedure is something that takes appropriate parameters and a current store and returns the new store. Intuitively what does the abstraction mean under such a semantic setting? Let us just consider this, let's just consider procedures, it means that the change from one store σ_1 to our final store σ_2 actually goes through a sequence of changes whose intermediate changes we are not interested in we are hiding them. You are only interested in what is the final store that is reached given this initial store. If you did not have a named procedure then what you would have had to do is you would have had to take that body C and place it inline in your main code in which case it means that you are essentially interested in every state that is produced by that body C . And the semantic abstraction that you provide by this is that you hide all the intermediate states while going from the initial state to the final state.

We will essentially look at it that way.

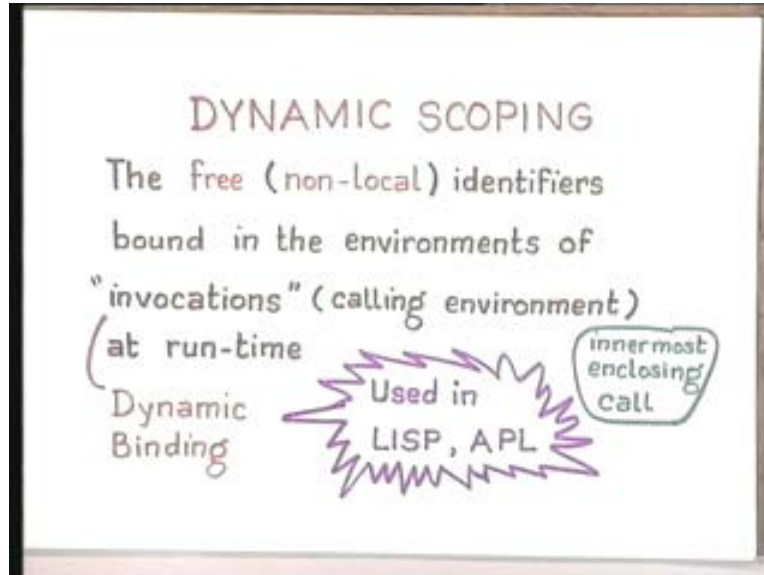
Now let us look at some other issues. Since we are talking about naming and identifiers we have to look at the issue of scope again. So the normal scope rules that we are all used to is what are known as the static scoping rules. In programming it is called a non-local identifier in a procedure or to be consistent with whatever we have done it is actually a free identifier of a procedure. So the free identifiers of a procedure are just those which are neither parameter nor have been locally declared. So every identifier that is neither locally declared nor which is a parameter by the way they are also declared is a free identifier or what might be called a non-local identifier and it is statically bound in the environment.

(Refer Slide Time: 38:25)



This means that the binding occurrences for such non-local references are determined statically. That means they are determined at compile time before any thought of execution of the program occurs and the binding occurrences are determined by the innermost enclosing scope rule. So, given any reference to an identifier, now this identifier also includes procedure identifiers, function identifiers and for anything which has a name it refers to the innermost lexically enclosing scope in which that name has been declared and that is something that is compile time determined. So this is known as static scoping or lexical scoping and the binding that is created by that is naturally called static binding.

(Refer Slide Time: 41:20)



So this is the binding that is created by the compiler as it reads through your code. Most languages including scheme important including scheme actually use static scoping rules. And as opposed to static scoping the other possibility is what is known as dynamic scope. Now what happens in the case of a dynamic scope is if you have dynamic scoping rules what it means is for any non-local reference the appropriate binding occurrence cannot be really determined at compile time and the dynamic means that whatever you do is most likely possible only at run time. Therefore whatever are the free or the non-local identifiers in the abstract they are bound not statically but they are bound in what might be called the calling environment. So what it means is that at different points in the program if there are different calls then at different calls the same identifier might refer to different things.

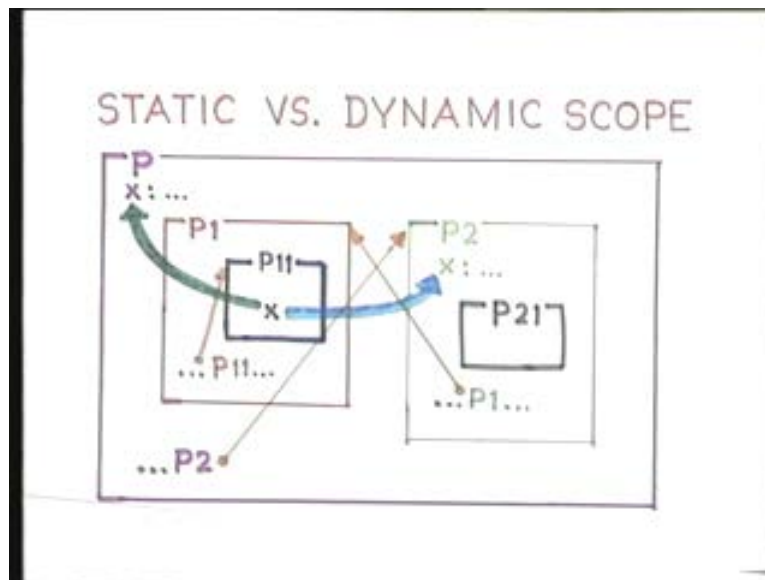
The binding occurrences cannot be determined by looking at the text of the program. The binding occurrences can only be determined by looking at the run time stack at that point. This is what is used in LISP and APL and what it follows is really the innermost enclosing call rule rather than the innermost enclosing block rule.

A block is a piece of text, a call is a run time object, there is a nesting of calls and the innermost enclosing call which actually contains a declaration for that object is the binding occurrence that you are looking for. So let us just look at these differences somewhat more pictorially. Actually I feel that all languages either have ML syntax or they have Pascal syntax but right now we don't even need to worry too much about syntax but we just have to worry about scope boundaries and so on.

Assume there is a program P which has a declaration of x. So when I write x colon dot dot dot it means that this is a declaration for x. And then within the program P there is a procedure P1 and within the procedure P1 there is a procedure P11 and this is what I might call the static structure of the program, the program as you read it on a print out,

there is no machine anywhere, just read the printout. If you read the printout and look at block boundaries then the program P is this entire (b...44:01) purple object and the procedure P1 is this red object lexically nested inside the program P, the procedure P11 is this dark blue object textually nested within the procedure P1 and after P1 there is another procedure P2 so you can assume that P2 follows P1 if you like so P2 and within P2 well there is a black procedure P21.

(Refer Slide Time: 44:00)



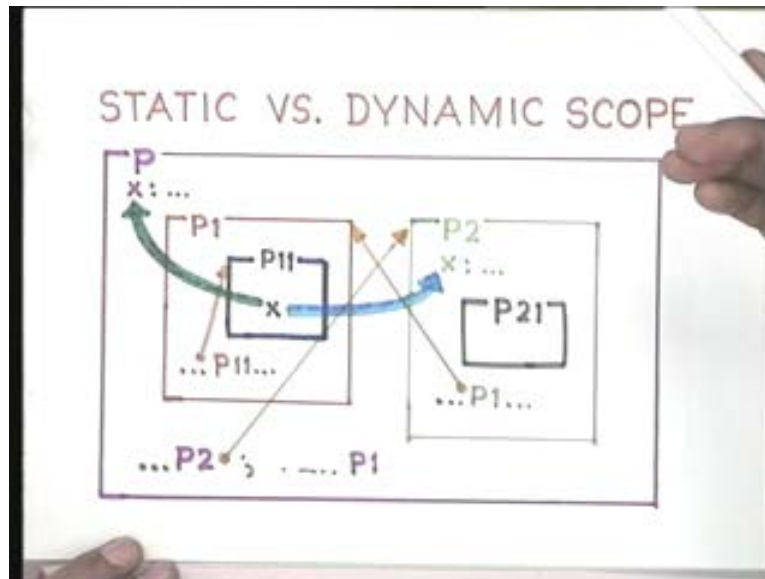
Therefore P2 contains a declaration for x. now the point is this. If this were a Pascal program and there is a reference to x inside this P11 what is the binding occurrence for that reference to x and that binding occurrence follows normal textual rules so you take the innermost textually enclosing block which contains a declaration for x. Therefore this x refers to this x. I am assuming that P1 does not have any declaration of x. Then the innermost enclosing block which contains the declaration of x is this purple x therefore this blue x is actually this purple x. So all references to x within this blue procedure are really references to this purple x assuming neither P1 nor P11 has a local declaration of x.

However, if this were not a Pascal program but a LISP or APL program then it is not clear what this x refers to. You cannot make out from the text of the program that this x refers to either this or this. What this x refers to really depends upon the execution time behavior. So let us look at an execution time behavior. Let us assume that this is a body of the main program P and there is a call to procedure P2 which means you have this green x here you are executing the body of procedure P2 and within it there is a call to procedure P1 and within P1 there is a call to procedure P11 and you find this reference to x.

Now the question is what does this x represent? What is the binding occurrence of this x? There is an x in the global environment because you called it from the global environment. There is an x also in an enclosing calling environment namely in P2. Now

which x does this refer to? If you use the innermost call rule then the closest declaration of x in the calling environment is this green x. Therefore this x refers to this P2. And the reason I am saying it is important to realize that it is the part of the calling environment is that sometime later in the main program if I had actually called P1 then assuming that it is a similar calling chain then P1 calls P11 and the same reference to this x now in the calling environment is of this purple x. Thus at different points depending on what the calling environment is the binding occurrences can change.

(Refer Slide Time: 48:19)



This means that at compile time you cannot make any commitments about the reference to this x. you do not know at all at compile time. At compile time what does it mean? You would have read the declarations of P, you would have read the declarations of P1, you would have read the declarations of P11, you will be processing the body of P11 and you encounter x and you cannot jump to the conclusion that since you have this x here it refers to this x here. Because at the point when you read this you have no notion what are the calls and what is the sequence of calls that are actually going to be performed at run time.

When you come down to the calls even if you chase the sequence of calls by that time this has become a black box which is not available to you, the information in P11 is no longer available to the compiler when you are processing the body of the program P. This means that in a dynamic scoping environment at compile time no commitments can be made about non-local references. And if no commitments can be made that means that the very act of compiling itself is actually a useless activity.

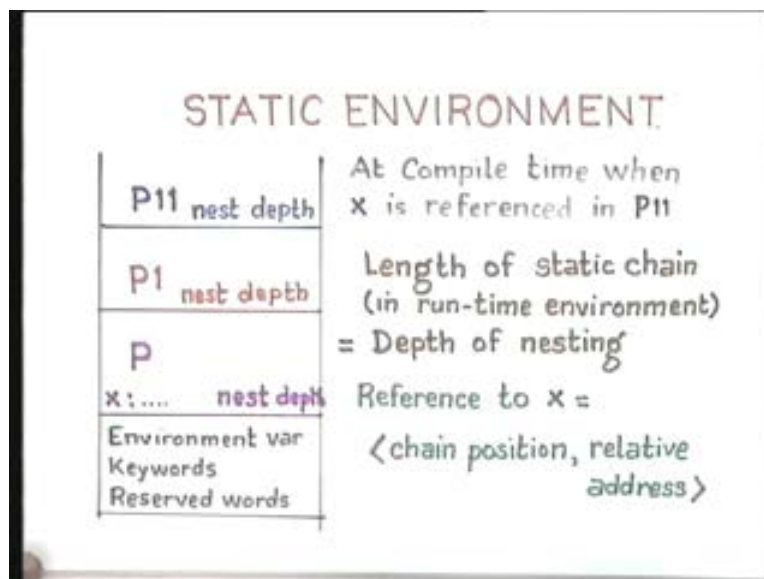
You cannot create any bindings, you cannot do any memory allocations because you don't know what x is, you don't know what x refers so you cannot do type checking, you cannot do memory allocation and you cannot do any storage representation so all of them have to be deferred postponed till you start executing the program which means you

might as well dispense the compiler and just interpret the program directly and that is why most LISP and APL systems are interpreters. They do have compiling features these days but the original LISP and APL systems were purely interpretive programming systems. And the reason for using dynamic scoping rules there was more pragmatic than planned.

This call that I looked at actually has, there might be an initial environment consisting of globals, libraries and so on which might be loaded then there is this call from P to P2, P2 to P1, P1 to P11 and there is a reference to x. And if you work backwards along the chain the innermost call rule tells you that this x under dynamic scoping rules refers to this green x whereas if you maintain somehow the static nesting structure of the program even in the calling environment then what you get is that you will have to go backwards along this static nesting chain and you will never hit this green chain x you will only hit this purple x. So, under static scoping rules it should give you the purple x for the same sequence of calls.

What it means is that static scoping though it is textually very nice and allows for debugging, allows for compiling, allows for compile time type checking, allows for compile time code generation and so on and so forth what it means is that there is an overhead associated with it. That overhead is that you have to maintain the static structure of the program. You have to capture the static structure of the program, the textual structure, the lexical scoping structure of the program somehow in your run time environment so that your references are statically determined. On the other hand, the dynamic scoping does not require this extra overhead of maintaining that information. You just traverse down the run time stack and take the first x that you can see the period and your dynamic scoping rules are implemented.

(Refer Slide Time: 54:21)



So dynamic scoping is not convenient for the purpose of reading a program or debugging it from a listing but it's a simple environment. At that point you take a decision to go down the stack and find the first x that you encounter and that's the x you are referring to whereas in a static scope what it means is that you have to maintain this information. After all static scoping is not preserved really in the run time structure because as you can see there is a call from P2 to P1 and P2 to P1 are at the same nesting level and they are independent. This means that your run time stack does not necessarily maintain all this textual information. Therefore, at compile time what you have for the same program is that at the time when you reach the blue x you actually have a static environment which includes type environment, nesting depths and so on and so forth with all this information in the symbol table not necessarily in this fashion.

Since the references of symbol tables are very frequent symbol tables are usually organized as hash tables through hashing. But the logical structure of the symbol table if you assume a linear search for references would be that you have this kind of a stack. There is a global environment of reserved words, keywords, environment variables, library identifiers and so on. Then you have the symbol table for your main program with type information, storage information and so on. Then as you go through you come to the declaration of P1 so you have the symbol table for the local declarations of P1 the parameters of P1 and so on and so forth and you keep incrementing and maintaining the nesting depth at each point then you will reach P11 you would have not touched P2 yet so there is no question of looking at the green declaration of x.

Therefore, what you look at in the static environment is you will just go down this logical stack because it is not really organized that way and find the first x and that gives you the innermost enclosing block which contains the binding occurrence. And if you maintain the nesting depths then the reference to any x is just to show how many nesting depths down you have to go followed by whatever relative address has been assigned to that x and that's the reference to x. So you can substitute all non-references to x by this ordered pair of nesting depth and relative address except that this nesting depth has to be maintained in the run time environment somehow and for that we have what is known as static chain point.