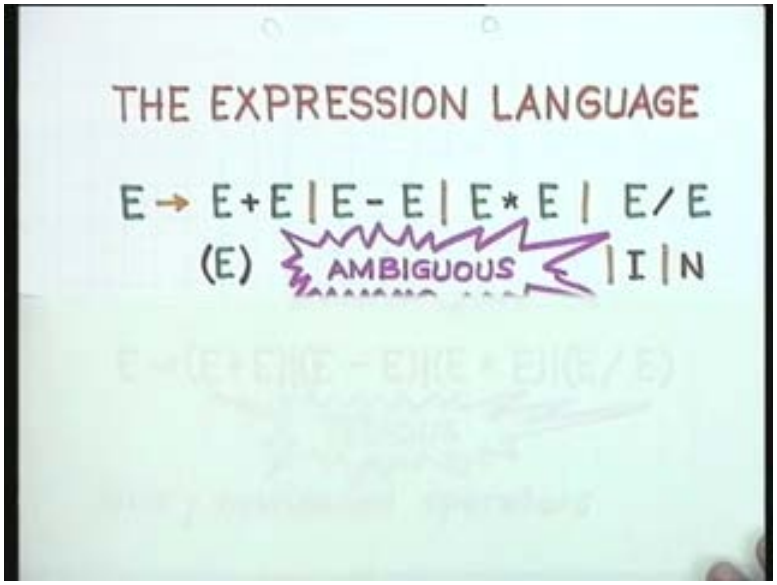
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 10
Lecture Title: Binding

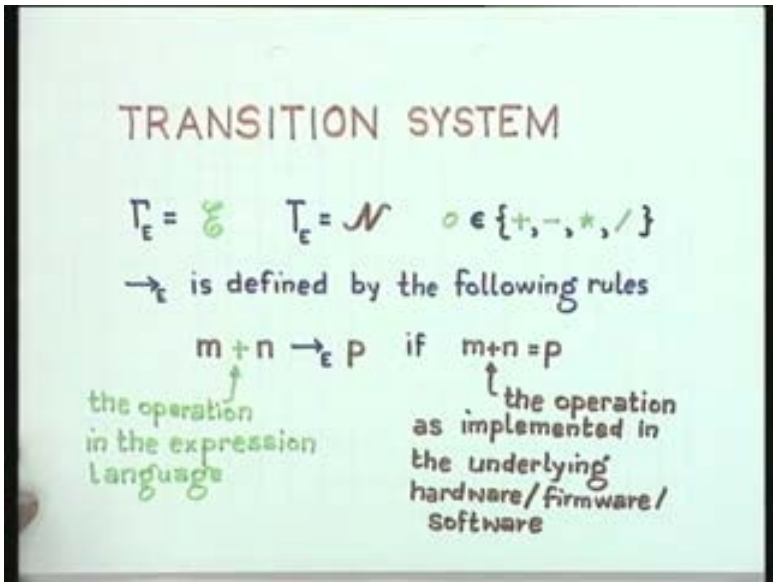
Welcome to lecture 10. Before we start this lecture on binding we will briefly recapitulate what we did last time. Last time I discussed transition system for simple expression language in which there were no identifiers.

[Refer Slide Time: 00:46]



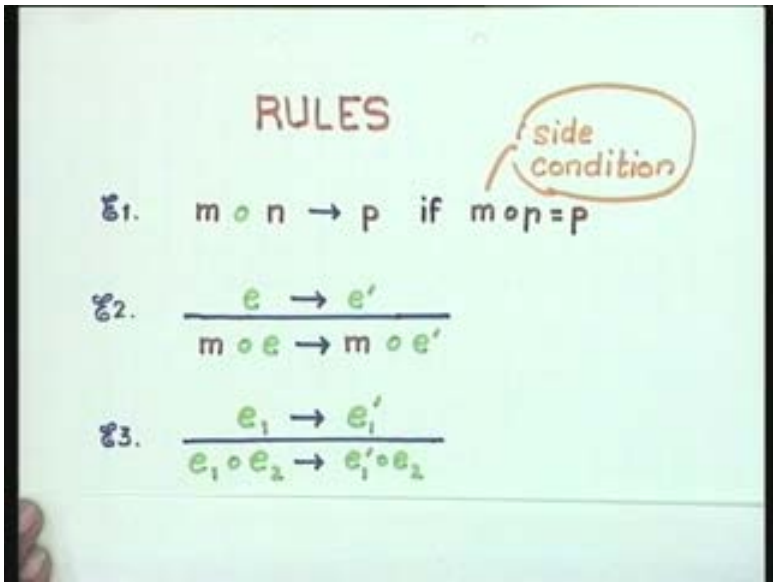
We would like to view expressions as trees and with that in mind we gave the following transition semantics.
We assumed a transition system, a set of abstract syntax trees of integer valued expressions of which there was an underlying set of numerals with notations for them. We defined a transition system in which the set of configurations was the set of all possible expressions in the language and the set of terminal symbols were the numerals.
I use this general symbol to denote any of these binary operators and we defined the transition relation by various rules. We had only three rules multiplied by the number of binary operators that we have.

[Refer Slide Time: 01:30]

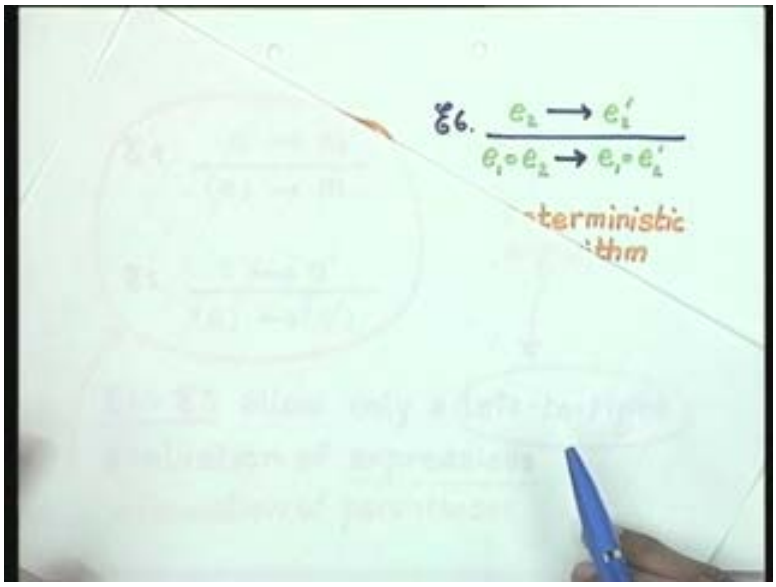


There are two numerals along with a binary operator under the understanding that the binary operator is directly implement-able by the virtual machine on which the expression language is implemented. It gives you another numeral and these two rules applied inductively essentially specify that we are allowing only left to right evaluations.
Given a binary operator you cannot evaluate the right operand of the binary operator till you have completely evaluated the left operand.
This rule applied several times will finally yield a left operand for the binary operator so that it becomes a numeral and then the right operand is evaluated. It is desirable in many cases to have a deterministic set of rules and in certain other cases it might be desirable to allow for a non deterministic set of rules.

[Refer Slide Time: 02:29]

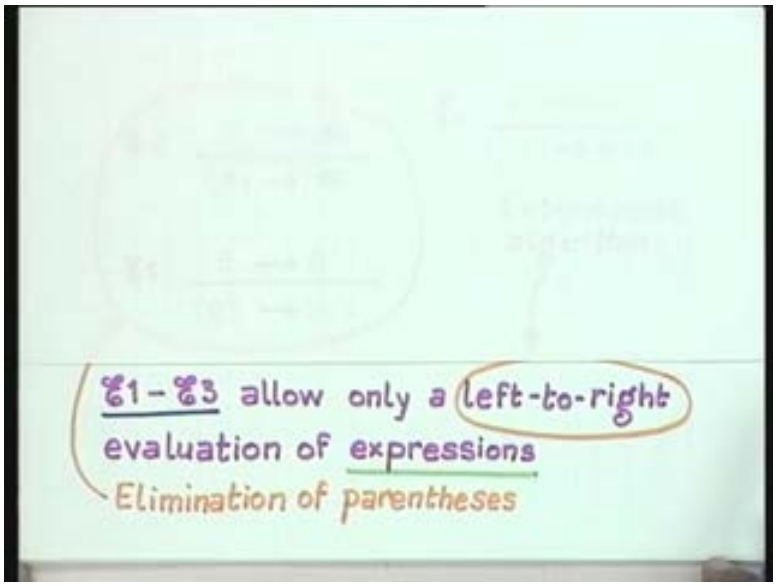


[Refer Slide Time: 03:19]



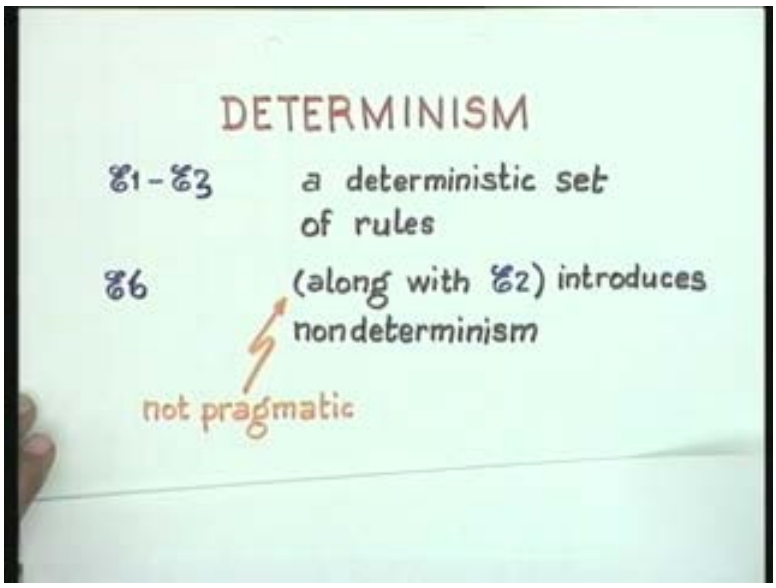
For example; if you were to allow the extra rule e6 then you get a non deterministic sequence of evaluation possibilities.
Otherwise if you allow only the first three rules then you only have a left to right evaluation of expressions and depending upon test you can add to your semantics whatever you require for parenthesis or you can eliminate parenthesis all together since you are only dealing with syntax trees.

[Refer Slide Time: 03:34]



Very often for pragmatic reasons we would like to have a deterministic set of rules and we will assume very often that we will give only a deterministic set of rules because that ensures that it defines a compiler for a uniprocessor quite unambiguously. Before we start on declarations it is necessary to give a general impression of what binding means in programming languages.

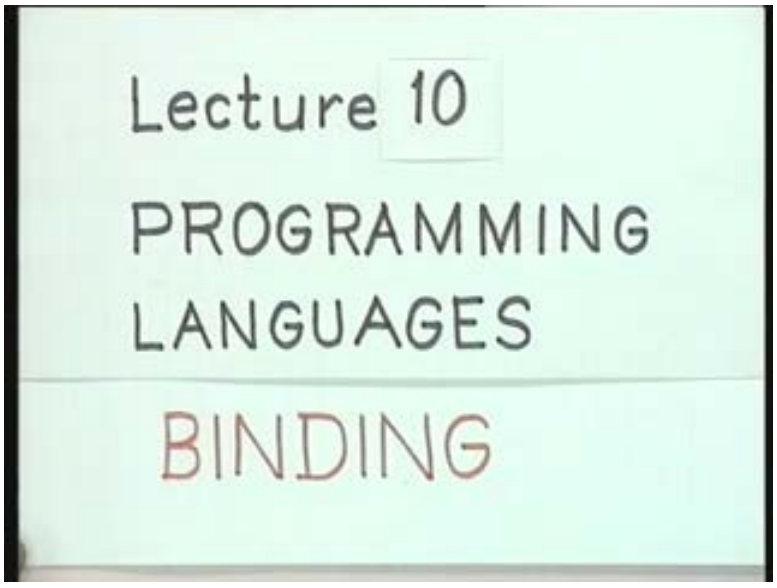
[Refer Slide Time: 04:00]



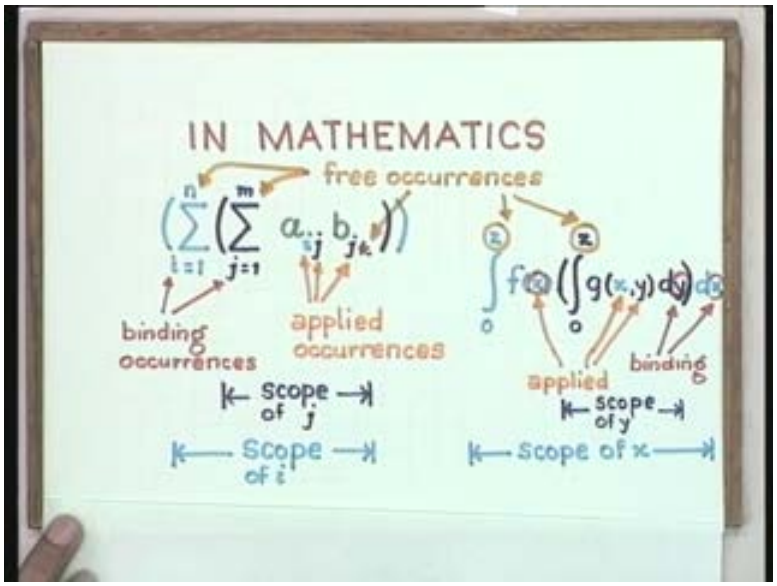
The field of programming languages is cluttered up with a lot of terminologies. Many of them come from implementation issues, many of them come from theoretical issues and many of them come because the terminology in different programming languages is

different. So, you require a unified terminology to discuss all programming languages. The notion of binding is often regarded as a purely pragmatic issue.
Let us now discuss the notions of binding that are prevalent for example, early binding, compile time binding, runtime binding etc. The earliest notion of binding occurs really in mathematics without mathematicians being consciously aware of it.

[Refer Slide Time: 05:00]



[Refer Slide Time: 10:30]



You have an expression like a double summation where you have names. Binding very often has to do with names and declarations. So, it is appropriate that before we discuss the semantics of declarations we discuss the notions of bindings but then we will discuss the pragmatic notions of binding too. Let us first concentrate on names.
There are the names ‘a and b’ but what matters to us at this moment are the names i, j, k, m and n.
We will assume that the summation in mathematics where j goes from 1 to m and i goes from 1 to n. The ‘i’ is a name and we normally use a name to denote some complex object. We use a name instead of having a long expression or a long phrase to denote some complex object especially when we are going to use it again and again to give it a name. Once having given it a name the issue that arises is what that name means at least in the context in which it appears.

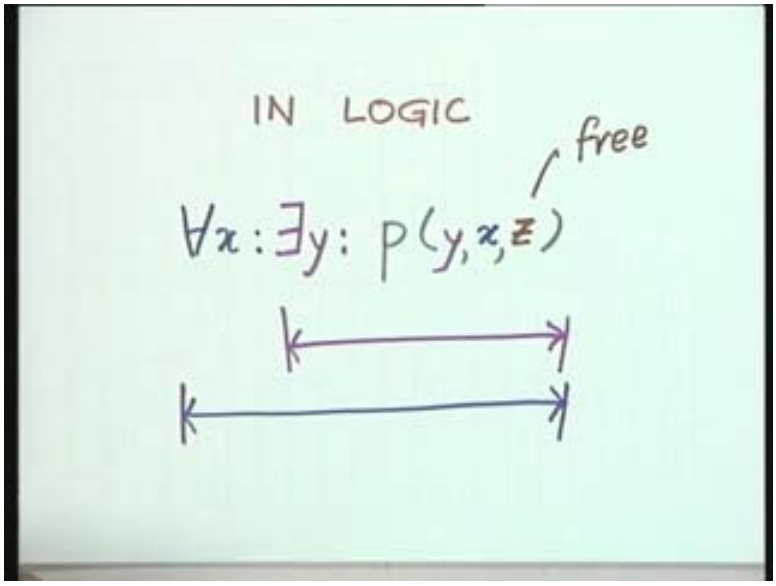
The meaning of the name is that ‘i’ goes from 1 to n which means that i can range from 1 to n and that is a binding occurrence. There is an introduction of a name along with its meaning. The entire sigma i = 1 to n has i as a binding occurrence and similarly with j in this $\sum$. Having introduced the name and what it denotes, I am now free to use that name for some purpose. Here ‘i’ denotes an applied occurrence so having defined a binding occurrence for the name 1 has an applied occurrence and it is similarly the case with j. However, k has a purely applied occurrence without any binding occurrence.
Presumably, this whole expression was lifted out of context and somewhere else. If it is a well defined mathematical context then k would have a binding occurrence somewhere else in the context.

Binding occurrences could occur somewhere fairly deep inside some mathematical object. Similarly n and m would presumably have some binding occurrences somewhere and they are called free occurrences. The whole issue has to do with names, naming, their meaning and when a name has a meaning.
Take a case where the binding occurrences precede the applied occurrences.
If you were to take a double integral, the applied occurrences actually occur before the binding occurrences.

For each of the bindings there is a scope. For example; the scope of j starts from its first binding occurrence to the end of the phrase which can have an applied occurrence.
The scope of i similarly goes to the end of a possible applied occurrence.
In the case of the double integral it starts essentially from an integral symbol and goes right up to the binding occurrence.

Even as a tree the binding occurrence occurs as a node of some right sub tree whereas the binding occurrence precedes the applied occurrence if it occurs as a node of a left sub tree. The definition of a binding occurrence is just that it is the first introduction which somehow gives it a meaning but meaning is not part of the syntax.
The meaning of a sigma and the definition of that sigma might differ. I will just assume that it refers to summation but that is not necessarily always valid. A binding occurrence usually means a declaration in programming.
Binding occurrences and applied occurrences occur everywhere in fact.

[Refer Slide Time: 14:12]



For example, in logic you have these quantifiers. You could have the following quantifier: for all x, there exists y and some statement in terms of some predicate p which probably has several free variables.

You could have z, x and you could have y. This is common and you have this binding occurrence of x here, an applied occurrence of x whatever this predicate is. The predicate is some statement $\forall x : \exists y : p(y, x, z)$.

Here you have a binding occurrence of y and you have one or more applied occurrences of y in the predicate p and the scope of x and y are delimited by the respective quantifiers and z is free. If you look at any of your school problems in mathematics one of the first statements you would say let x be something and then the rest of the context is probably a solution towards x or a solution involving x.

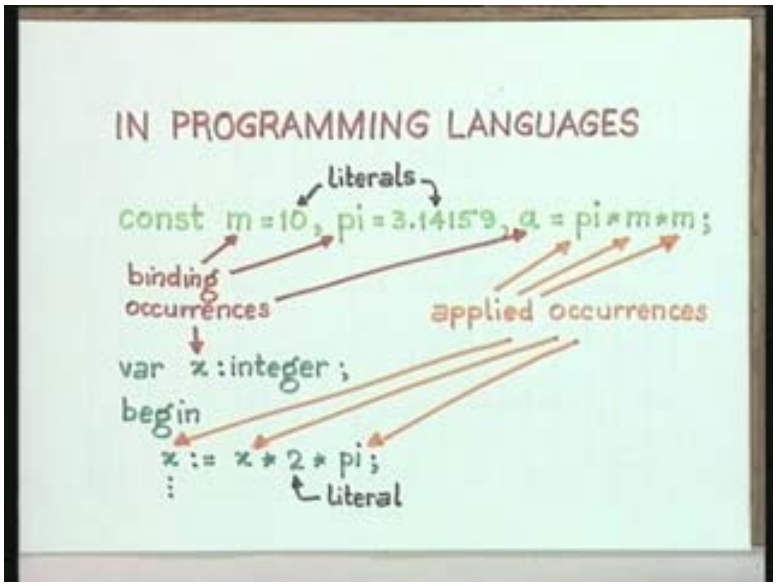
That statement ‘let x be something’ is a binding occurrence and all other occurrences which use that x are applied occurrences. In every problem you might have a statement ‘let x be something’ and ‘let y be something’ then in each case you state that you got a binding occurrence. In different problems the different ‘x’s have different binding occurrences and they therefore have different applied occurrences.

So, within a problem when you say let x denote something then that x has a scope restricted to your solution to that problem. This is very much part of natural language and to know where to define is very hard but to understand it is reasonably easy.

By first introducing a name and explaining it through some meaning, you give it a meaning and that is a binding occurrence. Later assuming that meaning, you can always use that name and every usage of that name is an applied occurrence.

Let us look at a typical Pascal kind of environment.

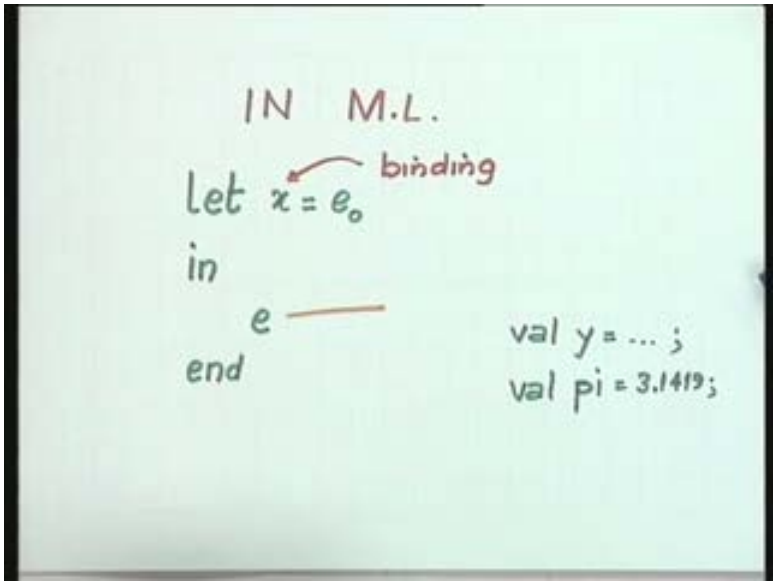
[Refer Slide Time: 18:16]



I have this constant declaration and in most cases in programming, binding occurrences occur as parts of declarations when you introduce new variables.
As in the case of mathematics the binding occurrence in several languages could actually occur after the use of that name.

For example; in mathematics it is also quite common to say let some expression be something where you have already introduced new names which are all applied occurrences and then you write where those new names have their meanings.
Those are the binding occurrences. For compiling reasons, many programming languages like Pascal for efficiency insist that binding occurrences should precede use or should precede applied occurrences.
Here is a constant declaration in a language like Pascal though I have used commas so you could equally well take it as a language PL 0.
Say ‘m’=10 where m is a constant, 10 is a literal, pi = “3.14159”, a = pi * m * m. In each of these cases the first introduction is a binding occurrence and all later uses of those names are applied occurrences. In the case of a variable declaration you have a binding occurrence then the variable is first introduced and everywhere that variable is used you have applied occurrences including on the left side of the assignment. All these are applied occurrences.

[Refer Slide Time: 20:00]

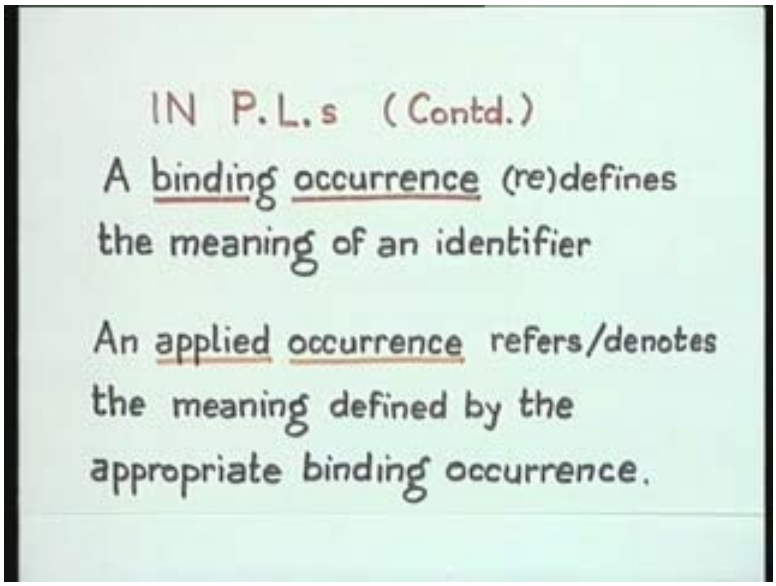


In a language like M.L., a ‘let’ statement is a typical occurrence of an introduction of a binding occurrence. “Let x = e0 (not) in ‘e’ end” where I will assume that this expression ‘e’ actually uses the x. There is a binding occurrence of x and any occurrences of x inside e would all be applied occurrences. In a typical ml session you also introduce names in the beginning as val y = ….; val pi = 3.149. All these are binding occurrences and you are creating an environment of names along with meanings and you are using those names in your subsequent expressions. The most important reason for having binding occurrences is usually the introductional names.

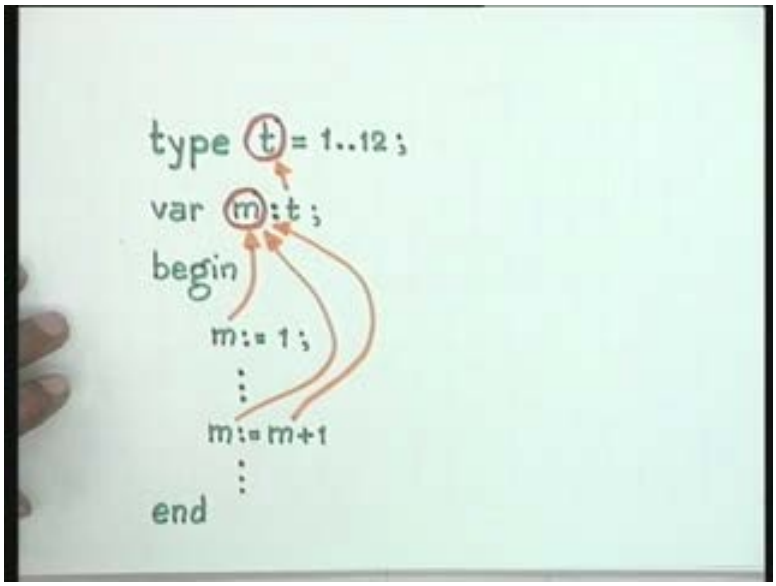
What are all the possibilities? A binding occurrence usually either defines by introduction or redefines an existing name. So, names are identifiers. Identifier is the more common term and an applied occurrence just refers to or denotes the meaning defined by the appropriate binding occurrence. In general the semantics of the language should actually specify what a binding occurrence is and what an applied occurrence is.

You could think of a declaration like this in Pascal as defining some binding occurrences and also applying them somewhere. Essentially we require names because we cannot always be using arrows. You could in theory completely get rid of names if you had some complex way of referring to some object or some meaning. In a language like Pascal which insists that all declarations precede use, the arrows will all be upwards. Further in a complete program of Pascal all variables will eventually be bound; all identifiers will be bound.

[Refer Slide Time: 20:55]



[Refer Slide time: 21:20]

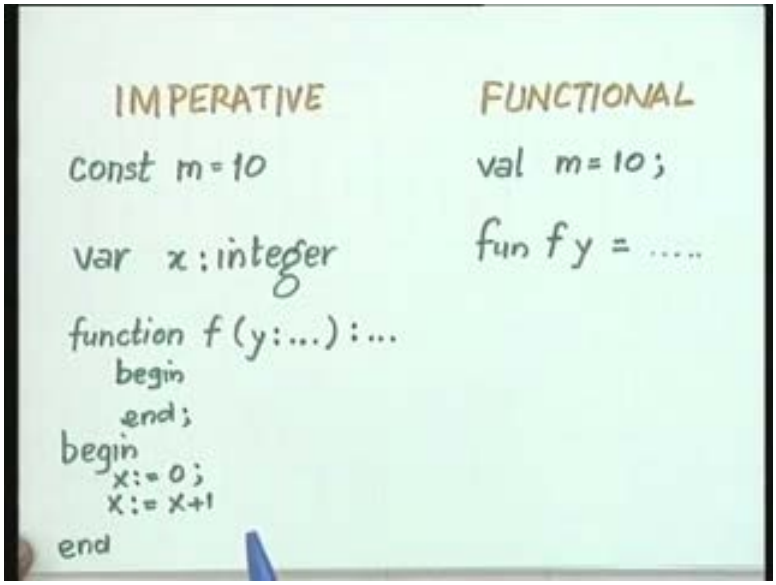


There will be no free identifiers and by a complete program of a language I also include the libraries which you might be using.
In a typical Pascal program or a FORTRAN program if you use the function ‘sin(x)’ that ‘sin’ has an applied occurrence in your program.
However, you might never actually define it within the program because it is already available as a library.

A typical environment of a FORTRAN program includes not just your program but also all the binding occurrences that are defined in the global environment through libraries or through the initial global environment that is provided by the system.
Let us look at the various kinds of bindings and let us look at the meanings of bindings more closely.

Let us just consider two simple instances. We will take a Pascal like and an ML like language. A Pascal like language has constant declarations say, ‘const m=10’ and has variable declarations of the form ‘var x: integer’.
An ML like language or a functional language has declarations of the form ‘val m = 10;’ and within ‘let’ expressions which are unnamed functions.
You also have function names for example; function f (y :…): … and you have expressions like this, ‘fun fy = ….’
In the case of an imperative language the constant declaration m gives you the binding occurrence of m which binds the name m to the value 10.
It is a name-value binding and in fact the functional language declaration of this form is exactly the same. It gives you a name value by name.

[Refer Slide Time: 30:14]



That is what binding means. The meaning of m is just the value 10 within the scope in which it is declared.
In imperative languages further there are these variable declarations which actually denote locations. The variables in an imperative language actually have name-location bindings. When you have a ‘begin end’ with the variable being updated then the variable-location binding remains unchanged however there is also a location-value binding in the case of a variable which can be changed by commands. In a functional language in general there are no locations. The whole idea of a functional language is that it is abstract enough not to include the notion of a memory as part of the language and therefore updates and assignments are prohibited.

A variable in a functional language is like a constant in an imperative language. A variable in a functional language is just a name-value binding and in languages in most cases you can also look upon the function itself as a value and a function declaration itself as another name-value binding.
Whereas variables in imperative languages implicitly assume the existence of a notion of memory or store and they actually consist of two bindings; a name-location binding, which is pragmatically speaking just a memory address.
There is a name-location binding and then there is a location-value binding which might be changed depending on what are the contents of the location or how you update that location.

In a functional language this complication of locations or memories is completely abstracted away and you have just name-value bindings throughout and that is the essential difference between a functional language and an imperative language.
What people consider a variable in mathematics is really a variable constant. It is variable in the sense that it is unknown but that its binding is the same. Over the scope in which you have a variable it represents the same value in a mathematical problem.
Similarly, in a functional language in a functional program over the scope in which a variable is declared, it represents only one value unlike an imperative program where the variable actually gets updated.

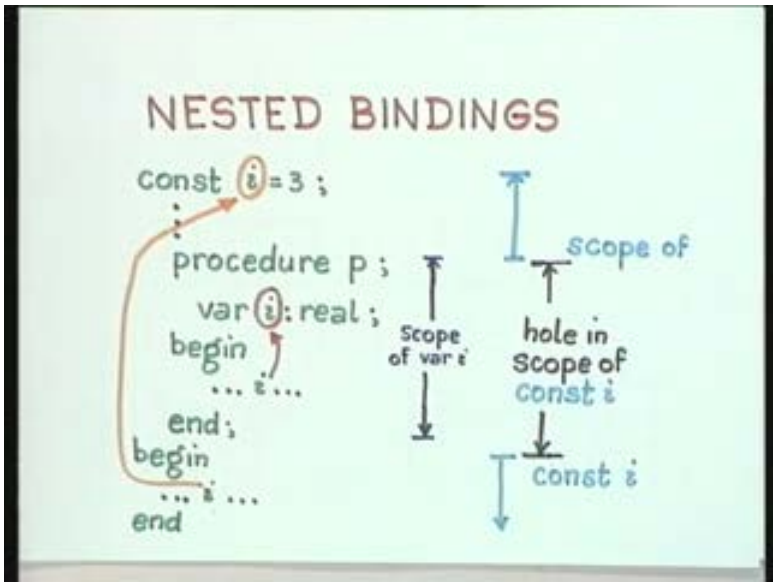
Only the name-location binding remains constant. The location-value binding keeps changing constantly. The notion of a variable in a functional language is really the notion of a variable in mathematics which really does not vary over time. It is a variable in the sense that in mathematics usually the name variable is used as something that is unknown but something that has a constant value. In a functional language since it is just a name-value binding it is always constant over the scope in which the binding is effective as opposed to a variable in an imperative language where what is constant is only the binding between the name and the location or the address in which that variable is stored.

The name x just denotes a particular address in memory pragmatically speaking and that alone remains constant over the scope in which this binding is effective. It does not say anything about the location-value bindings that loosely speaking, in the case of memory locations, means the contents of that memory location which can keep changing.
Since the issue of bindings came up it was necessary to point this out but we are really more interested in bindings from the point of view of declarations because we eventually have to give the semantics of declarations of let us say some language like PL0.

Let us look at the various kinds of bindings that are possible. You could have nested bindings. For example; in a typical Pascal like program fragment let us suppose you have a declaration of the identifier i and you have another declaration of i as a variable. All applied occurrences of i within the ‘begin end’ block refer or mean whatever name-location binding has been specified. All occurrences of i denote whatever name-value binding has been specified.

In the normal lexical scope rules the scope of the variable ‘i’ declared extends through a procedure. The scope of the constant ‘i’ extends through the entire fragment provided there is no other declaration of i in an inner scope and any other declaration of i is a hole in the scope of the constant i. So, in the case of what are known as statically scoped languages, a scope just gives you an extent in the program text over which a binding occurrence applies.

[Refer Slide Time: 32:10]



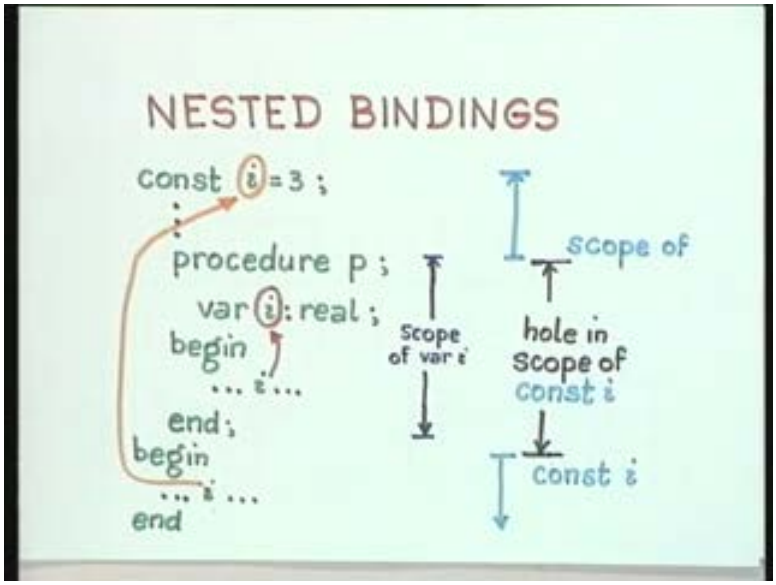
Also as a language designer the one question that arises is ‘are nested bindings necessary?’ Why does a language like Pascal which for example boasts of simplicity a good language for learning programming and why do almost all block-structured languages allow nested bindings which can only be confusing?
For example; they affect readability because every time there is a reference in the main core to an identifier, you have to find the declaration where the actual binding appears and if you allow nested scoping and the creation of holes in the scope of a given identifier, then you are spoiling the readability of a program.

Secondly, it also makes it necessary for a compiler or a parser, every time there is an applied occurrence to look for the appropriate binding occurrence and verify various aspects. For example; within a scope are you using let us say, ‘i’ only as a real variable or as a character or a string or are you using it as a pointer? It puts an extra overhead on the compiler and the runtime system especially in a language like Pascal which does type checking as to where exactly the binding occurrence is and whether the meaning specified by the binding occurrence is being explicitly followed.

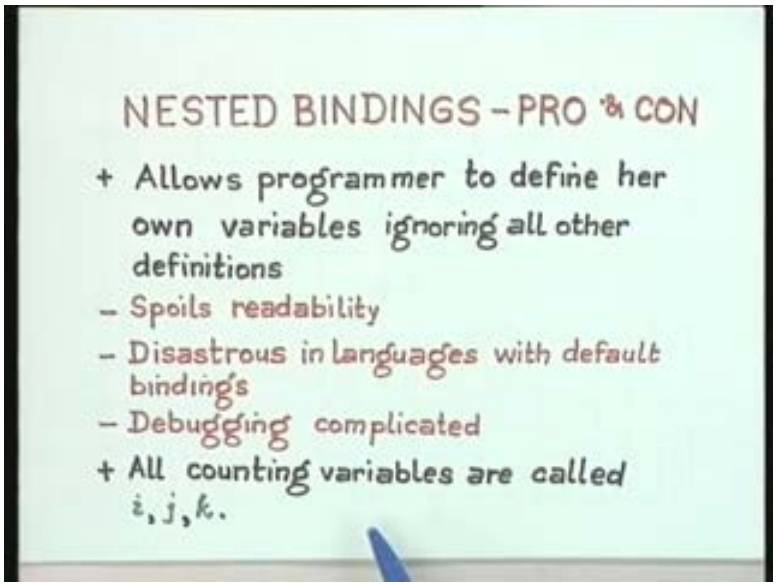
A simple solution could be to just ban all nested bindings so that every time you declare a new scope you have only new names and there should be no problem at all.

But very often when you have a program developed by a team what you would like to give a programmer of an individual procedure, which might go deep into some program is the flexibility to use his own names for the variables that he is using. This means that if you disallow nested bindings then that programmer would have to know all the global names before deciding what should be the new names to be used locally within that procedure, which can be a difficult task.
The entire programming team should first decide what the names are for various global variables. In a really large software project that can be a constraint rather than facility. However, it spoils readability; it is quite disastrous in languages with default bindings and it complicates debugging.

[Refer Slide Time: 38:23]



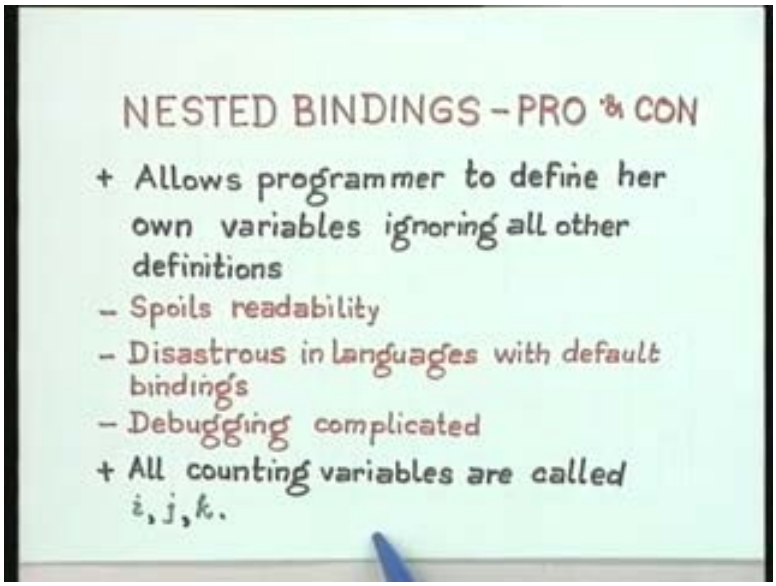
[Refer Slide Time: 38:44]



For example; if you intended to introduce the name in your scope but you actually forgot to declare it the debugging process becomes quite complicated because every time that name is considered to be global. In this case if you were the writer of the procedure and you forgot to introduce the declaration then it will be taken that the name always refers to some global occurrence and you will be wondering why your program does not seem to work as you expect. It really complicates the matter and the name may not be in just one outer scope; it might be global and your procedure might be deeply nested inside somewhere.

So, nested bindings complicate matters quite a bit both in terms of debugging and readability but what really clinched the issue in the case of Ada, for example is that everybody uses i, j and k as counting variables in a for loop.

[Refer Slide Time: 38:58]



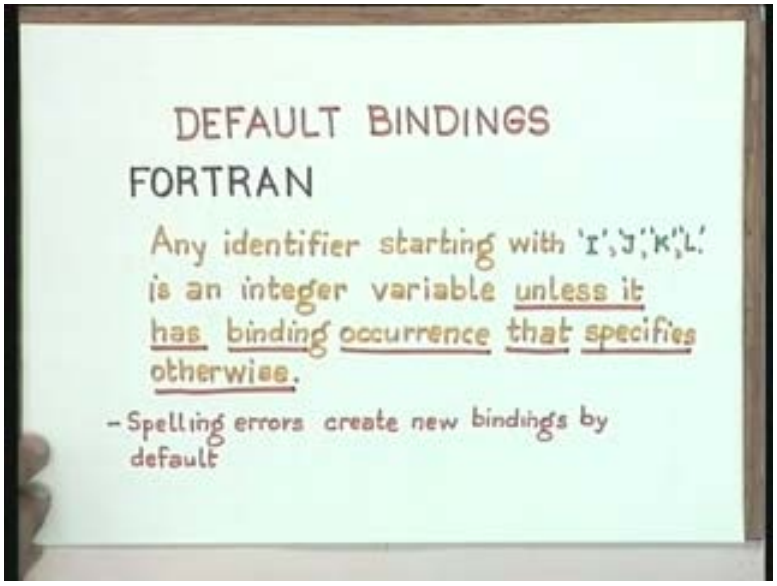
It was actually stated in a discussion on the Ada language that everybody in the world uses i, j and k as counting variables. So, you should give the programmer the flexibility to decide especially when the counting variable has no other significance except as a counter for a 'for loop' statement for example. All of us use just i, j and k if it has no particular significance. Nested bindings are here to stay even though they complicate some matters.

In languages like FORTRAN for example, they had a different rule.
They said that you do not need to have a declaration which means that you need not have binding occurrences; you need to have only applied occurrences.
However, the applied occurrences will have default bindings in this fashion. Any name that starts with i, j, k and l which does not have a binding occurrence is taken to be an integer variable.
Further any identifier that starts with any other letter and which does not have a binding occurrence is taken to be a real variable.
They had these conditions which actually make matters really bad because what can happen without these binding occurrences is that you choose your identifiers so that they reflect the problem context.

You might actually have something starting with i; you intended it to be a real variable but you forgot to declare it and the FORTRAN compiler also compiled it, the runtime system is executing it but you are getting disastrous results and you may never know about it. You may never know about it especially if you are solving a problem for which there are no known test results. There is a legend that some of the space disasters of one of the satellites was due to the fact that it was written in FORTRAN where there was
a 'do' statement in which a comma was mistakenly replaced by a period and therefore that entire 'do' statement was taken as a real variable.

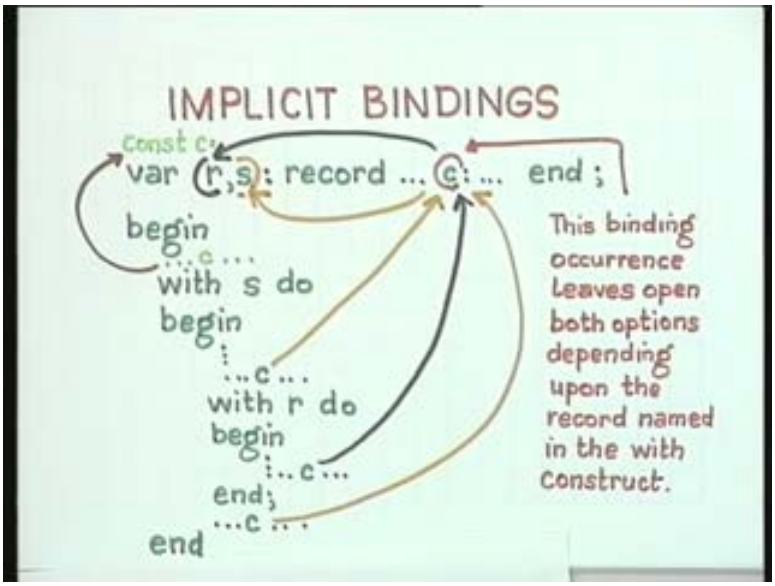
So, ‘do 10 i = 1,15’ means you execute that loop 15 times but if the comma is replaced by a dot then this ‘do 10 I’ is taken as a real variable since it starts with d and there is no binding occurrence with an initial value of 1.15.

[Refer Slide Time: 42:39]



There is a legend that one of the Venus disasters was due to a FORTRAN program bug which definitely is never detected by a compiler; it is definitely not detected by a runtime system because there are no binding occurrences due to the default bindings.
What it means is that small typographical errors and spelling errors which we are making all the time in our programs would get bound by default to something other than what we intended them to be. That is one of the reasons why later languages like Algol 60 insisted that there should be binding occurrences for every new identifier.

[Refer Slide Time: 44:05]



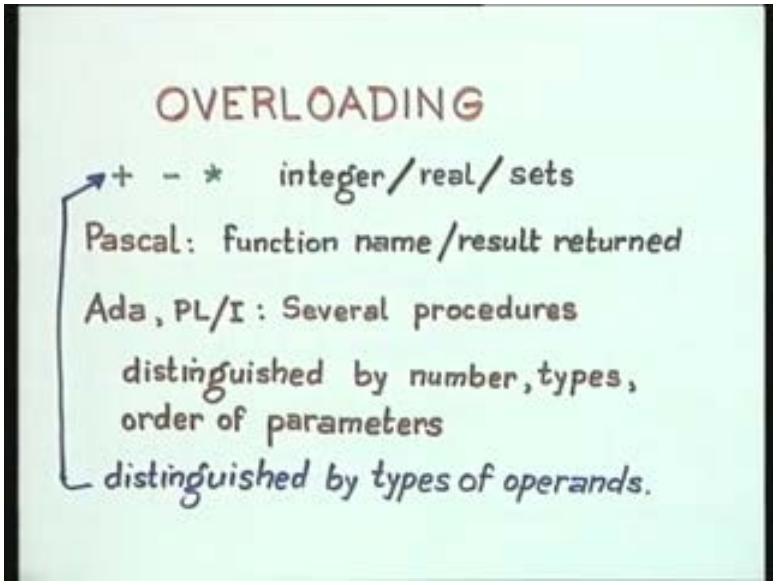
There are certain bindings which are implicit for example; a constant declaration is not really intended to be in a scope, it could be in some outer scope. You could have two records say, ‘r and s’ in a Pascal like language.
Let us say c is a component of a record declaration which has some type and c has a binding occurrence within the record but however c is only bound to all variables having that record and is of a particular type. So, c is not explicitly bound either to r or s both of which are of a record type.
The binding occurrence can actually be stretched out or decentralized. Part of the binding occurrence can be in one statement, the other part of the binding occurrence can be in a ‘with’ statement. Within the ‘with’ statement for example, c actually gets perfectly bound to r. Within the ‘with’ statement all references to c actually get bound to the s and the normal scope rules apply.

The ‘begin end’ is a scope; ‘with s do begin end’ is another scope and ‘with r do begin end’ is a hole in the scope of the scene which is bound to s.
Pragmatically speaking, even binding occurrences can actually be stretched out but the full meaning need not appear at a single binding occurrence. It might be distributed across 2 or more constructs which is actually not very surprising in the case of variables. There the name-value binding occurs at one go and the location-value binding occurs several times in different places.

Even the name-location binding in the case of variables does not fully occur at that point in many cases. If you look at the implementation of a language like Pascal, a name relative-location binding occurs at compile time and relative location to absolute-location binding occurs at runtime because the structure of a block structured language is such that you never know at compile time when you are using a dynamic memory allocation, when you are going to call that procedure and what is going to be the base address relative to which that variable has to be located.

Even when you look at the name-location binding purely pragmatically, it can be stretched out across the entire spectrum from compile time to runtime.
There are other kinds of bindings. For example; in all kinds of binding that we have discussed so far you have an identifier denoting a single object whatever that object might be but you have also what is known as overloading and the most common overloading available in all programming languages is the overloading of addition, multiplication and subtraction.

[Refer Slide Time: 48:59]



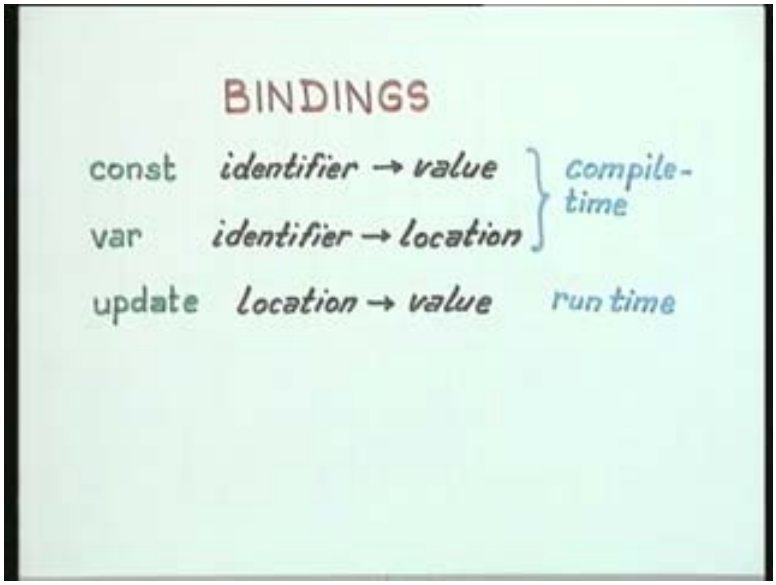
The operators are also identifiers and their binding occurrences are in the global environment in most high level programming languages.
However, even in the global environment they have two different meanings depending upon the operands. There are actually two different identifiers, integer addition and real addition or floating-point addition, integer multiplication and floating point multiplication. Within the same scope these identifiers have two different bindings simultaneously available. So, there is no creation; there is no overriding of one binding by another, the overriding is completely local and it depends entirely on the types of the operands. This is known as an overloading.

In the case of Pascal, another obvious overloading is that you have a function name and a variable with the same name which stores the result of executing the function. So, that is a case where you have a function name which denotes a function object but the same name within the same scope also denotes a name-value binding in which the value of the function is returned. Besides, the overloading has been carried across more modern languages also much further.
For example; in languages like Ada you can have the same name bindings representing different objects which are distinguished just by something simple like the types of parameters.

In the case of procedures; within the same scope you can define two or more infix plus operations for example; you might do plus for matrixes of some kind, you might do some plus to denote concatenation of strings etc.

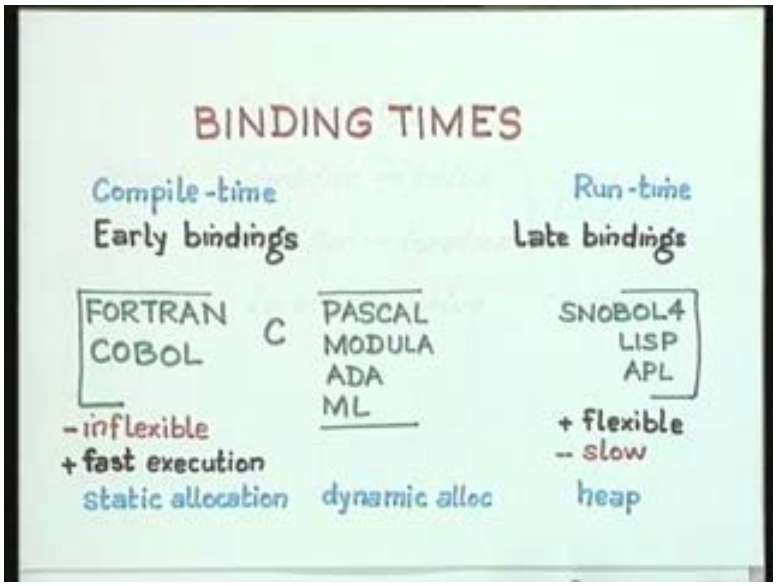
In addition to the global bindings of plus which denote integer addition or real addition you have two more new bindings and all four of them occur simultaneously in the same scope and the only distinction is either based on the types of the operands or the types of parameters which is mathematically the same though in terms of language implementation, it is different. It depends on the types of operands or order of operands. Just on those syntactic bases you have a distinguishing capability between different bindings for the same name available within the same scope.
Lastly, let us look at the various kinds of bindings from an implementation view point.

[Refer Slide Time: 50:40]



You can have constant variable bindings where some of them are compile time and some of them are runtime mostly in languages which believe in a static type checking or at least which allow the facility of static type checking.
You try to do the bindings as early as possible at compile time. In many other languages you actually do the bindings at runtime languages. Even the identifier value and identifier location- bindings are often done at runtime.
This is typically with dynamic data structures. Even in Pascal you have various name-location bindings occurring at runtime.
The language, Lisp for example carries it much farther so languages like SNOBOL and Lisp believe in runtime bindings as opposed to compile time bindings.

[Refer Slide Time: 55:37]



They believe in what are known as late bindings. As a result you can take the entire spectrum of doing bindings and you can actually, loosely speaking, order languages.
FORTRAN and COBOL do very early bindings. Everything is done at compile time. The complete memory allocation is done at compile time.
Even if the FORTRAN program has re-locatable code, the bindings are completed.
For example; the bindings of variables such as name memory address, absolute address bindings are done at loading time before runtime begins.
A compile FORTRAN program might still give you a re-locatable code that means it might only give you relative addresses but during linking and loading the entire address calculation is completed for every variable.

For every name the address calculation is completed before. As a result FORTRAN also has a static memory allocation. Since you do very early binding you have to decide on exactly the kind of memory allocation well before you start running the program.
A static memory allocation is done very much like in the case of assembly language. Every code segment is followed by a data segment for that code and so you have very early bindings well before runtime.
There are no allocations done at runtime. You have a completely statically partitioned memory and all absolute addresses are calculated well before the execution of the program. At the most they are delayed till loading time.

So, FORTRAN has this property and as a result FORTRAN programs are very fast because the overhead of memory allocation and changing bindings is not there.
But they are also quite inflexible in various ways. You cannot have dynamic arrays.
You cannot have dynamically created data structures, so, at the cost of flexibility you have very fast executions. In SNOBOL and Lisp since the bindings are all late and most of them are done at runtime, you have to do fresh memory allocation.

You have to do all the bindings like name-address bindings at runtime which means that you also require a garbage collector which tells you exactly what part of the memory is being used, what part of the memory is not being used and the request for more memory that are all done at runtime. As a result these languages are in general very slow but they are highly flexible. You can do the data structuring completely dynamically with absolutely no static allocation. Most of the allocation is on the heap in such languages. Languages like Pascal, Modula, Ada and ML actually follow an in-between policy. They do a name-relative address binding at compile time but they postpone the relative absolute-address binding to runtime.

In the case of C, actually most of the allocation and mostly static allocation is done as in the case of FORTRAN except where the C compiler detects recursion. It detects recursion by a graph construction process and the moment it detects recursion it realizes that you really cannot do the static allocation that you do in the FORTRAN because you do not know how many recursive activations of that function are required. You need to do fresh allocations each time you activate. So, whatever is not recursive follows a policy of FORTRAN and only for recursive invocations it takes a late-binding view and actually there is a graph construction process by which it detects cycles in the graph and therefore it also detects recursion.

C programs are very fast because they do most of the bindings at compile time.
In the case of Pascal, Modula, Ada and ML there is an extra overhead in the form of type checking which has its own overhead.
At least in the case of Pascal it is just type checking; in the case of ML it is type inferencing which means there is a huge computational process involved in just compiling a function. Since you do not need to specify types in the function there is a type inferencing system which actually solves the system of equations on types and so there is an extra overhead at compilation.
In the next lecture we will start off with actual declarations in PL 0.