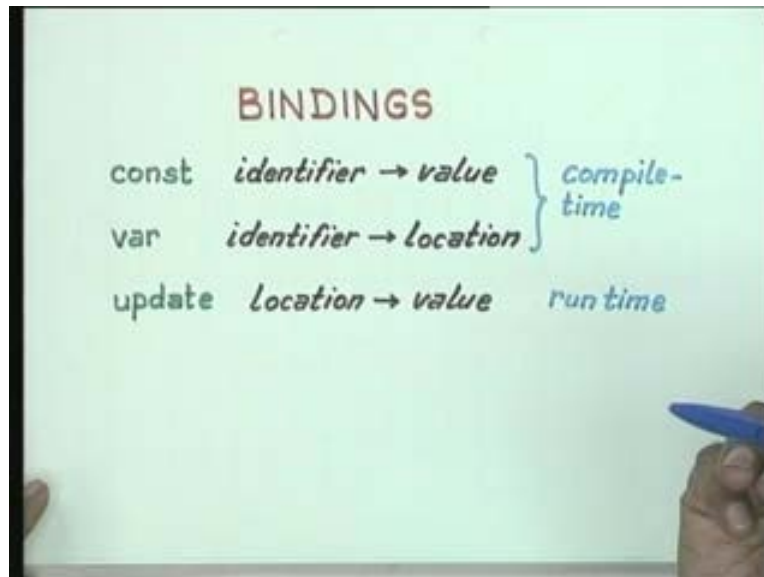


Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 11
Lecture Title: Environments

Welcome to lecture 11. We will talk about environments today after briefly recapitulating what we did in the last two lectures. In the last lecture we discussed the notion of bindings. In general we are talking about identifier-value bindings in the case of constants, identifier-location bindings in the case of variables and in the case of updations, normally assignments; we are talking about location-value bindings. I also said that pragmatically speaking you could decide to stretch out a binding for example; you could actually split up an identifier-location binding into an identifier-relative location binding and then relative location to absolute-location binding and do it in a sort of temporarily decentralized manner and depending on the time at which you do various kinds of bindings you might either have early bindings, late bindings or somewhere in between.

Bindings by themselves are a pragmatic matter in the sense that they really depend upon the implementation but there is a certain core of bindings which is not pragmatic but a semantic matter and that is just the fact that there is something known as a constant and that a constant declaration is somehow different from a variable declaration. That is a semantic matter.

(Refer Slide Time: 02:29)

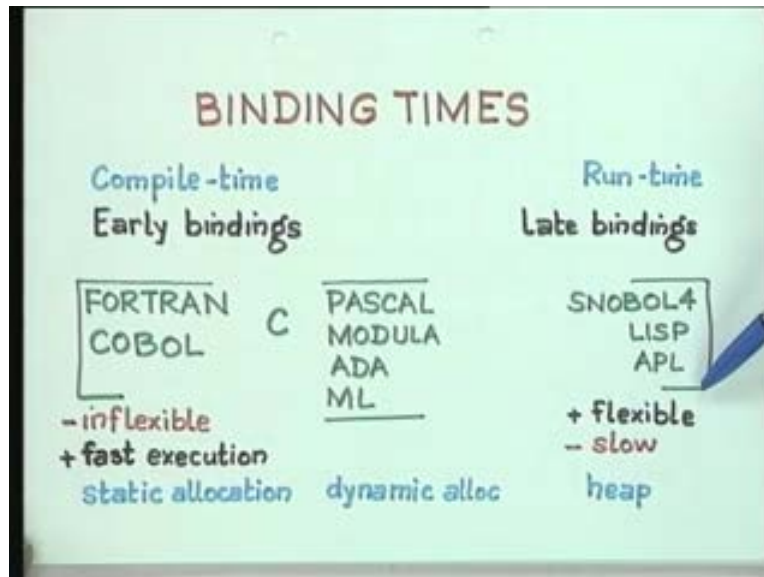


How you actually do the bindings is pragmatic. For example; if you take a language like PASCAL, let us say, type checking in PASCAL there is absolutely no guarantee. Typing of an identifier is also a binding. There is absolutely nothing in PASCAL in the definition of the PASCAL language which says that you should do type checking only at compile time. In fact there is nothing about the language which really specifies that you should actually only design a compiler. You might as well design an interpreter if it suits your convenience. However, the language is being designed in such a way as to make compile time checking easy.

Most PASCAL compilers would try to do as much of the type checking at compile time so that the overheads do not run into execution time. So, it is a pragmatic issue in the sense that you want to make executions as fast as possible whereas it is not the case with languages like SNOBOL and Lisp which want to produce extreme flexibility. So, they delay all bindings right up to the end and it is all done dynamically but it is still a pragmatic matter. There is absolutely no reason why you cannot do some bindings earlier. Let us look at a semantic specification of certain kinds of bindings especially the

creation of environments, the processing of declarations and how they affect the rest of the language.

(Refer Slide Time: 04:02)



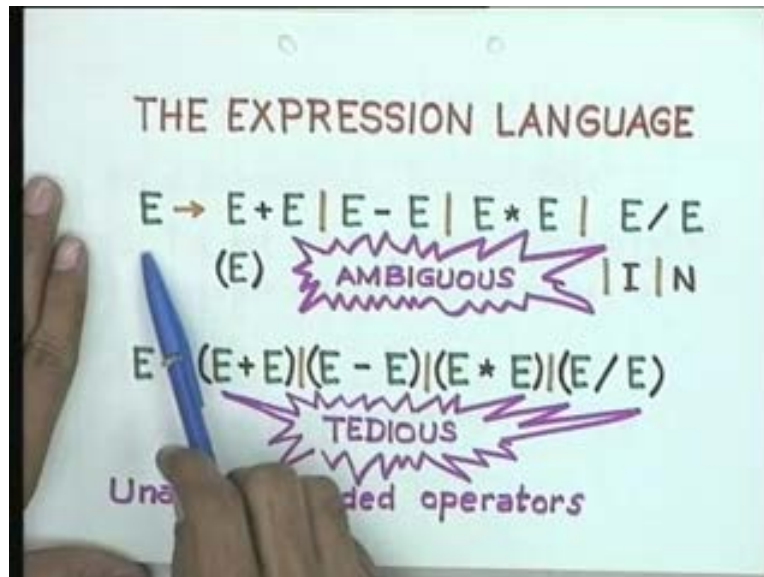
Let us briefly go through the previous lecture. We had defined an expression language of this kind in which I had explicitly forbidden identifiers. It was a pure expression language consisting of just numbers. So this production 'I' was absent and the reason is that an identifier has a meaning only if it has got some binding. It means that we can talk about the meaning of an expression containing identifiers only after we have obtained a meaning of the identifier in terms of some bindings. In this case it is an identifier-value binding that we are looking at. So, the expression does not have a value. The meaning in our case mostly represents the ultimate value that an expression is going to have and the expression does not have a value till the identifier has a value.

Let us look at the language of expressions but in a slightly more holistic fashion.

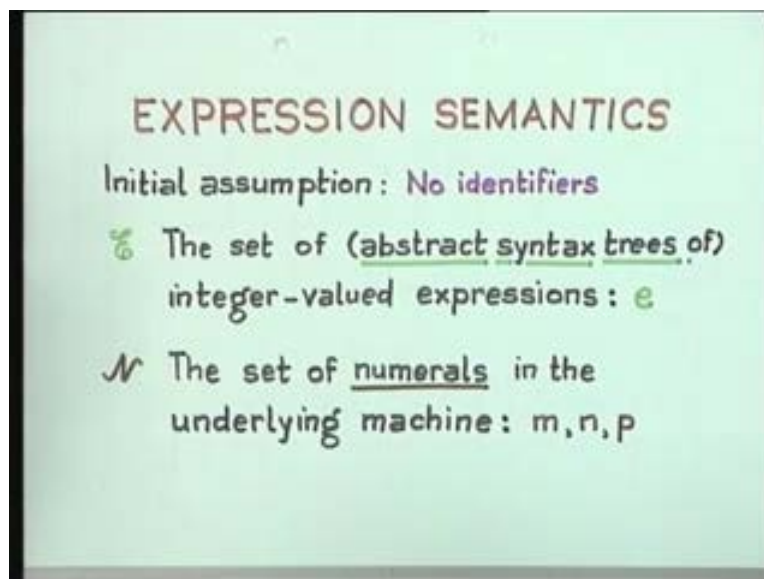
I assume that there is a set of expressions capital script E and a set of numerals script N. Script E is the set of all integer valued expressions E. The capital E denotes the non terminal symbols of the grammar. The small e denotes the syntax trees. This means that there is some processing which has already occurred if you want to look at it

pragmatically. I am really looking at syntax trees rather than the actual syntax of the expression which means that there is no ambiguity.

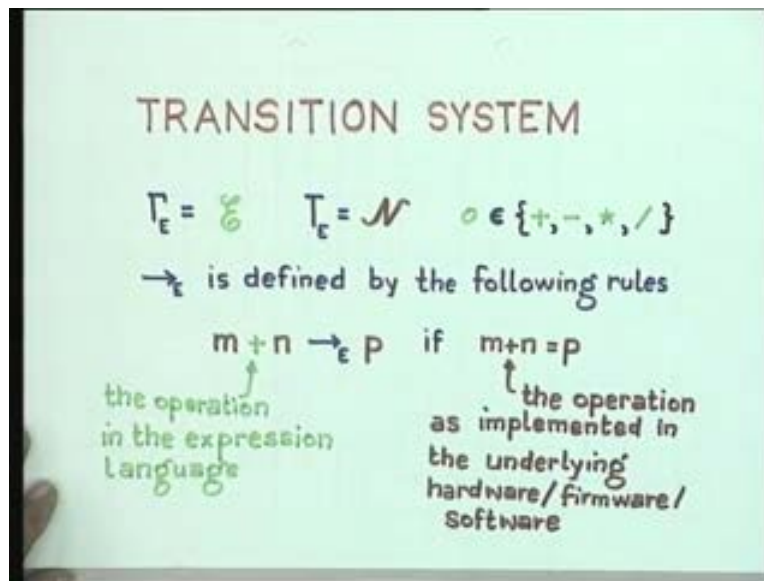
(Refer Slide Time: 05:40)



(Refer Slide Time: 06:06)

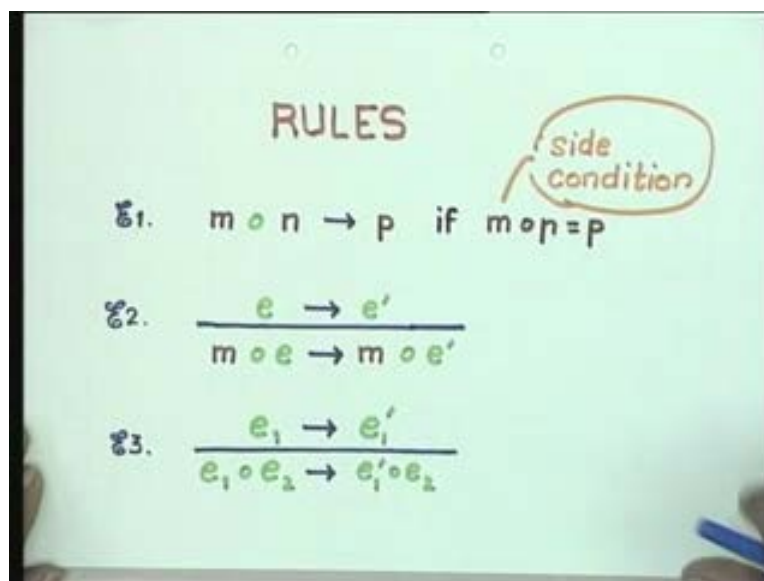


(Refer Slide Time: 07:23)



Similarly, \mathcal{N} is the set of all numerals in the underlying machine and we defined a transition system with the set of all configurations being the set of all expressions and the terminal configurations being the set of numerals which are a subset of the set of expressions. We defined the three rules which actually give us a left-right evaluation of expressions.

(Refer Slide Time: 07:34)



In an expression language which contains identifiers we will take a simple language with declarations very similar to what is obtained in ML. In actual ML, if this 'i' identifier has got a value E then you would write 'Let I = E in E end'. But since we are not dealing with anything else except values I have removed the keyword 'val'. The language of expressions just consists of $E \rightarrow N \mid I \mid (E \text{ o } E)$ and an expression of the form, Let I = E in E end where of course there is a declaration hidden inside that expression I = E means that there is a binding occurrence of that identifier and that declaration is itself expressed in terms of an expression E and if you have a binding occurrence then you have a scope for that binding occurrence which is given by a complete expression.

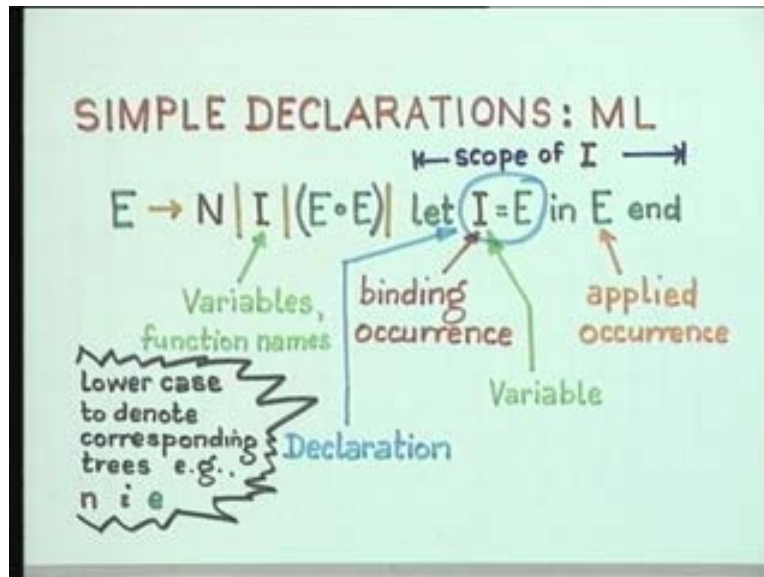
The identifier 'I' might occur in the final expression E as an applied occurrence. Informally, we are saying that the value of the expression E is the meaning that we want ultimately. However, the expression is too complicated, so, we have used an identifier inside that expression and the meaning of that identifier is somehow syntactically represented by the declaration I = E. The meaning of the entire expression is really the meaning of E which might contain possible occurrences of the identifier 'I' that has been declared. The identifier 'I' of course has a value which is given by another expression.

You could simplify the conditions by replacing all occurrences of the identifier I by the expression E and you would get a simple expression in the language. However, as you must have seen through some of our ML programs, it is always a good idea to allow for such declarations and such naming because very often they have a meaning that is associated with the problem domain and instead of writing one long monolithic expression you would split up the expression into various parts that are logically connected with the solution of the problem.

The identifiers here in ML could actually be variables in function names but for the present we will just consider variables. With ML being a functional programming language, the notion of a variable is really that of an identifier-value binding and the reason I am not going directly into PL 0 declarations is because the constant declarations in PL 0 are very much like the variables in ML. They both share the same kind of binding characteristic and the same semantic characteristic with that of a variable in an ML

program which is just a value and a constant in a PL 0 program or in a PASCAL program. We will look at the whole expression language.

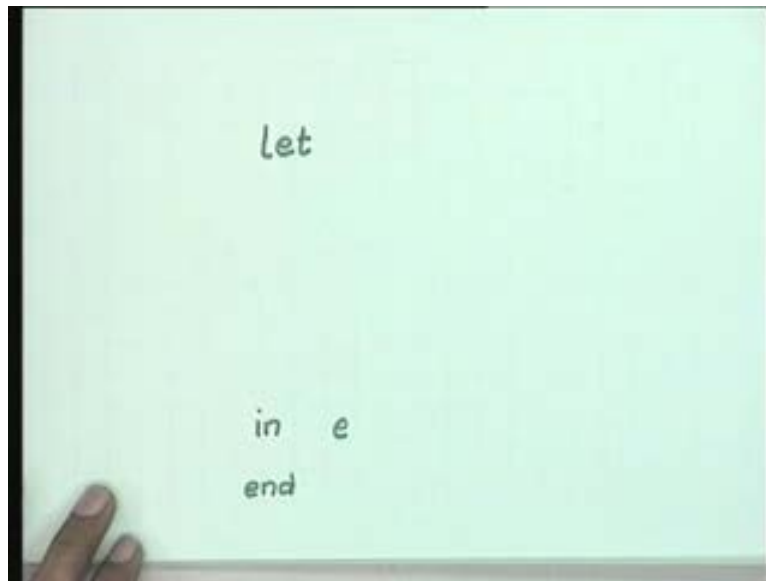
(Refer Slide Time: 12:00)



What is an ML program? An ML program or an ML like program is really just one expression. If you look at the ML programs, you have a collection of functions of values and declarations. Then you have a function which is what is compiled and an expression whose value is evaluated with those functions. You can think of any ML program as consisting of a 'Let' with a huge number of declarations and finally an expression which has to be evaluated.

In the case of an ML session the outermost 'let in end' is considered implicit and it is hidden and you do not have to explicitly write it. But you should think of any ML program or any functional language in the same holes and scheme too. In any functional language you can think of the entire program as being enclosed in a huge 'let in end' and since it is an interactive program the outermost 'let in' and 'end' are considered implicit and after having compiled all the declarations you just give an expression to find a value, but each call to an expression is really like having an entire program which is with all those declarations enclosed in a 'let in end' statement.

(Refer Slide Time: 14:03)

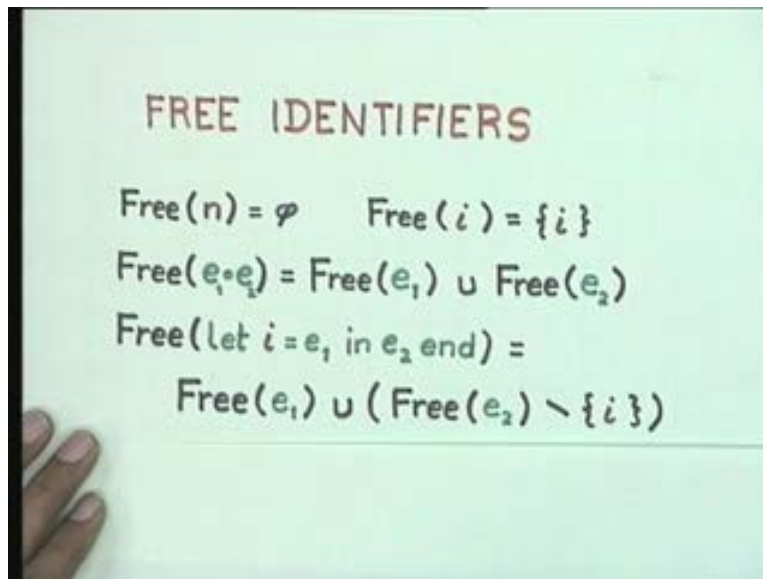


Let us look at the notion of identifiers. As any area of mathematics or logic we have to define when identifiers are free. The notion of binding means that an identifier gets bound to something otherwise it is free. Free here means that given a certain expression or a sentence of a language the free identifiers are those which do not have any declarations in that sentence. That sentence is supposed to be part of a larger sentence in which those free identifiers will get a binding.

Given a sentence a free identifier is one which has only applied occurrences and no binding occurrences. We can define the notion of freeness of identifiers by looking at the structure of the syntax tree and we can define it by induction on the syntax tree. Let us look at our grammar for expressions and follow the syntax of the grammar as closely as possible.

Given any numeral the set of free identifiers in it is empty. The set of free identifiers is a set of names that occur in it without a declaration. Given any identifier 'i' which is also an expression it is the singleton set of identifiers, {i}. Given any expression of the form (E o E) with this little 'o' being the root operator of the syntax tree, the free identifiers in the expression is just the union of the free identifiers in the individual components. If you have a let construct then the actual expression value that you are interested in is e_2 and the free identifiers of the entire expression $\text{Free}(\text{let } i = e_1 \text{ in } e_2 \text{ end}) =$ where a 'let' construct is a complete expression, are just the set of all free identifiers in e_1 since you are using probably some identifiers in e_1 to define the identifier i. It is a set of all free identifiers in e_1 and the 'i' will presumably be used in e_2 which was the whole purpose of having the declaration of 'i'. However, any occurrence of i here is an applied occurrence which has a binding occurrence $i = e_1$ and so, i is not a free identifier of the entire expression. The set of free identifiers in the entire expression is really the set of all free identifiers in e_1 e_2 excluding 'i'.

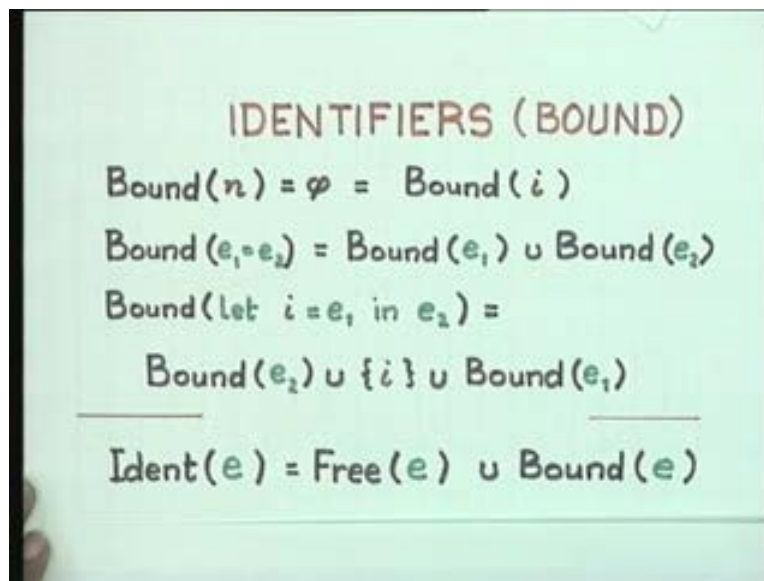
(Refer Slide Time: 18:26)



Similarly, we could define the set of bound identifiers and it follows a similar pattern. The set of bound identifiers is also defined by induction on the structure of the syntax tree. We must note that a simple identifier is free.

The numerals and simple identifiers do not have any bound identifiers in them and the set of bound identifiers in an expression is just the union of the set of bound identifiers in the individual components. Note however that two expressions could have the same identifier bound in different ways. You could have an x inside e_1 with the declaration for x ; you could have an x inside e_2 with the declaration for x and the two x 's need not be the same.

(Refer Slide Time: 20:20)



IDENTIFIERS (BOUND)

$$\text{Bound}(n) = \varnothing = \text{Bound}(i)$$

$$\text{Bound}(e_1 = e_2) = \text{Bound}(e_1) \cup \text{Bound}(e_2)$$

$$\text{Bound}(\text{let } i = e_1 \text{ in } e_2) =$$

$$\text{Bound}(e_2) \cup \{i\} \cup \text{Bound}(e_1)$$

$$\text{Ident}(e) = \text{Free}(e) \cup \text{Bound}(e)$$

Bound identifier is just the set of all bound identifiers in e_2 union the set of all bound identifiers in e_1 and in e_2 any occurrence of ' i ' would not be considered bound and so that has to rebound separately. The set of all identifiers of an expression is just the set of all free and the set of all bound identifiers in it ' $\text{Ident}(e) = \text{Free}(e) \cup \text{Bound}(e)$ '. The notion of bound identifiers has to do with already possessing a meaning which in turn means that the expression can be readily evaluated. The notion of freeness and bound identifiers exist in all programming languages either through implicit bindings or explicit declarations.

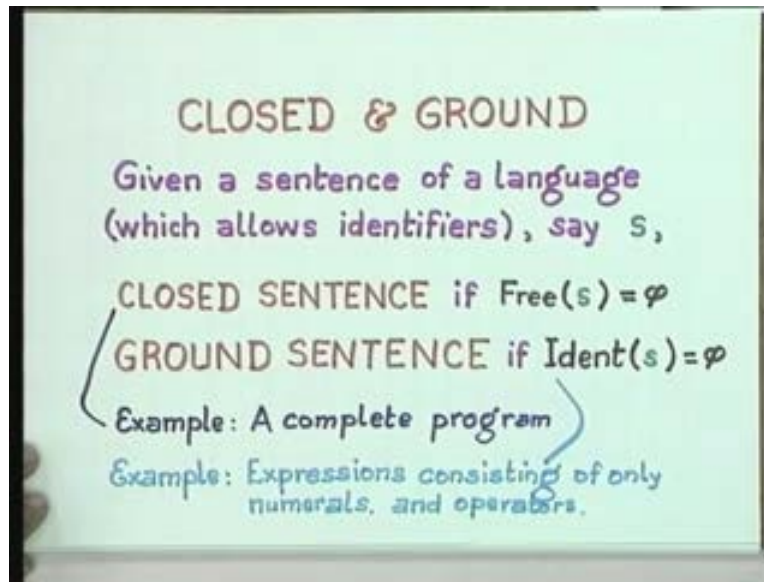
If you look at a complete Pascal program there are no free identifiers except those which come from the global environment of the Pascal run time system. All those library routines which are readily available are the identifiers that come from the global

environment of a PASCAL runtime system. Supposing you do not use any of the library routines then the complete PASCAL program has absolutely no free identifiers. The program heading has a name and that is itself a defining or a binding occurrence of that name of that identifier. Otherwise, if you are not using any of the library routines, every identifier is declared before use. If you look at the entire PASCAL program there are absolutely no free identifiers. All identifiers are bound which means that every identifier in that PASCAL program has a binding occurrence which defines it completely. A complete program of the kind which has absolutely no free identifiers is called closed.

In any language which allows identifiers, you can talk of a closed sentence of the language as one which has absolutely no free identifiers. A ground sentence is one which has absolutely no identifiers at all. In fact the transition system we consider for the expression language without any identifiers consists of only ground expressions. ‘Closed’ and ‘ground’ are two important terms which occur in any formal language.

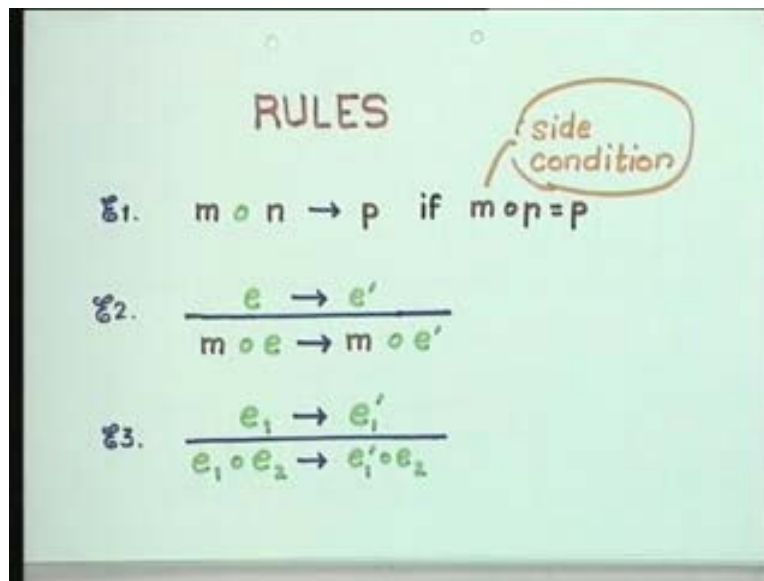
If you look at a complete Pascal program, since there are no free identifiers the program itself can be executed in an empty environment because there are no identifiers which need to be given a meaning from outside. A free identifier is one which gets its meaning from outside the sentence that you are currently looking at. In a complete program since all identifiers are bound they get their meaning from within that sentence and therefore a complete program executes in an empty environment.

(Refer Slide Time: 24:40)



A ground sentence does not require an environment at all because there are no identifiers which require bindings. Our transition system for expressions actually gives us a transition system in an empty environment. Since we consider only ground expressions without any identifiers they obtain their meaning in an empty environment. For instance I cannot give you the value of $(AX)^2 BX + C$ unless you give me a binding for A B C and X. Given that A, B, C and X have different values the expression has a meaning in the respective environment. So, that environment is just the name value bindings for A B C and X. It is just a set of name to value equalities.

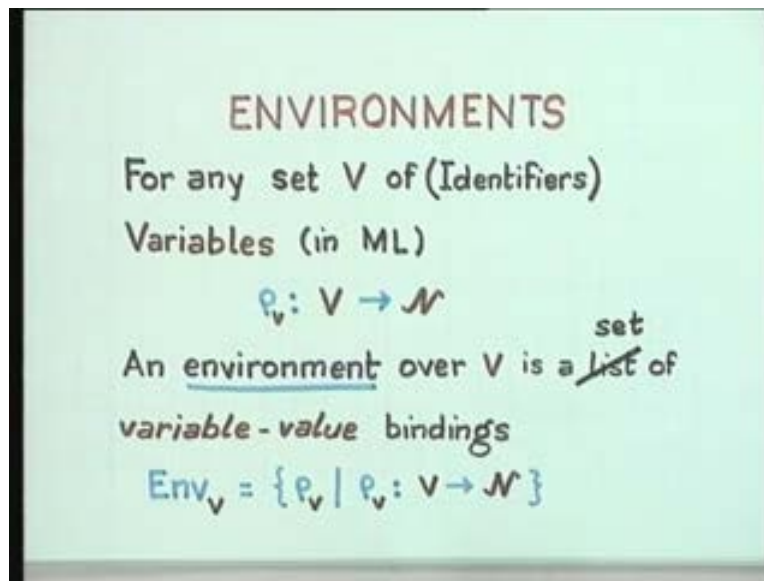
(Refer Slide Time: 25:06)



We will define the notion of an environment for this simple ML like expression language given a set V of identifiers. They are like the variables in ML. An environment row over the set V is just a mapping from capital V to the set of numerals. You can think of an environment as actually a list or more accurately a set of values. With a set of variable-value bindings and for any given collection of variables, V we can look at all the environments that are possible. I will use ENV_V to denote the set of all environments over capital V . Very often the subscript V will be omitted when it is either understood or it is not important. Otherwise, we are always considering an environment with the given set of identifiers and that set of identifiers could be extendable.

Let us look at the declaration language. We could actually extend that language further. Since declarations have a semantic meaning completely different that is they are logically a different entity from expressions, it is a good idea to abstract that out as a separate non terminal. In fact by abstracting it out as a separate non terminal the language of PL0 also gives some separate syntax and separate reserved words to make it clear that it is semantically a different logical entity from that of expressions or commands.

(Refer Slide Time: 27:38)



Once you have this there is absolutely no reason why you should have only one identifier declared there. You could have several identifiers declared. You have a separate declaration language whose basis is a single declaration and you can recur on that basis and give sequences of declarations which happens in most programming languages.

We have a functional programming language whose main non terminal is the language of expressions and there is an auxiliary language within it which is the language of declarations. Let us look at the dependencies. The expression language allows for declarations inside expressions and declarations in turn allow for expressions inside declarations and they both allow for identifiers. As you can see there is circularity in the entire definition when you apply all these, inductively or recursively.

Let us look at some quick examples. A typical sequential binding could be of the form 'let $\pi = 3.141$; $2\pi = 2.0 * \pi$ '.

The semicolon is very important in the sense that in a sequential declaration what you are sequencing might be important.

You are explicitly saying that the second declaration might actually depend on definitions given by the first declaration. Very often dependence is not there in which case they could actually be written in either order. This is also typical in PASCAL, for example; you could have a constant declaration of π ;... and another constant which uses the previous declaration of π in the evaluation of its value. You could have a complete expression in a constant declaration provided all the identifiers in that expression already have been declared before. In the case of PASCAL of course they should all be constants too; they cannot be variables.

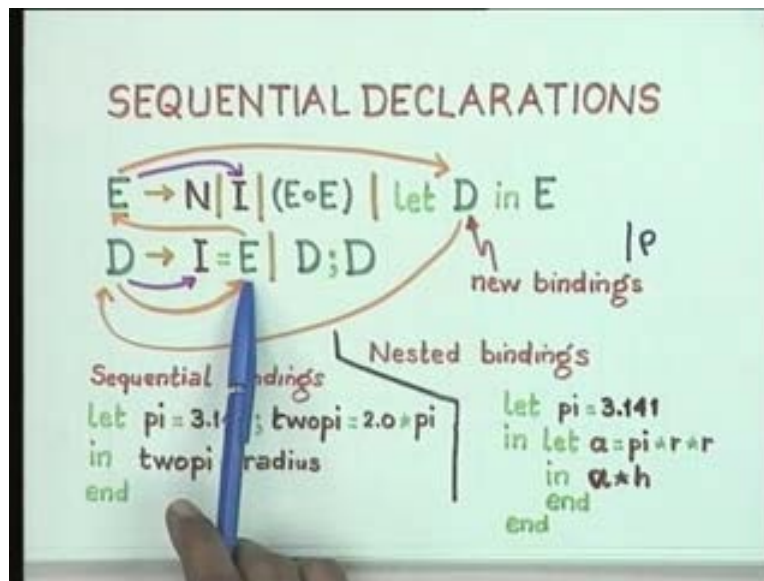
The identifier r is free in the entire expression and presumably the expression has been pulled out of some context in which the r has a binding but it is an expression in itself. So, let us assume semantically that in order for an expression to have a value which is what we mean by the meaning of an expression it is necessary that there be an environment in which the r has a value and once that is done an expression can be evaluated.

You can have sequential bindings and you could also have nested bindings. Let us say that the actual expression you want is $a * h$ where h is free and so the value of ' $a * h$ ' is what you want. However, since the expression presumably was getting too complicated you had to define ' a ' and in order to define ' a ' you had to define π . We have two occurrences of the identifier r in $\text{let } a = \pi * r * r$ which are free and so, it is not possible to evaluate the entire expression;

'Let $\pi = 3.141$, in let $a = \pi * r * r$, in $a * h$ end' unless r and h declared somewhere in the context are given some value from some external environment. You could have nested bindings too and it is in fact allowed in ML. By nested bindings you are really localizing the identifier. It is local only to the scope between let in and end.

The identifier π is local entirely to the scope.

(Refer Slide Time: 35:10)



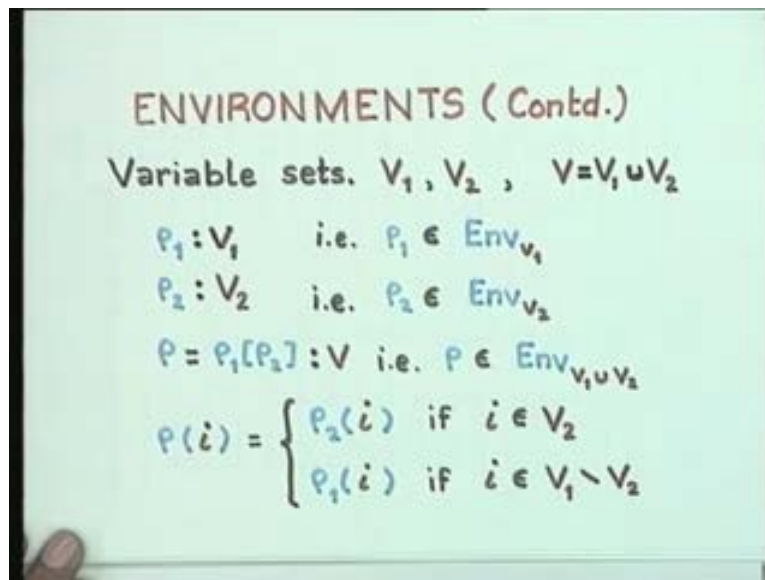
Such a construction is possible because an expression can have a declaration in it, a declaration can have an expression in it but if an expression is a full fledged expression then there is absolutely no reason why it cannot have a declaration inside it. We have to keep in mind the fact that they are possible when we define the semantics of the language.

We have defined the localization of new environments in the example we have taken. Let us define an operation on environments. Assume you have got $V = V_1 \cup V_2$. Note that I am not assuming that V_1 and V_2 are necessarily disjoint. V_1 and V_2 could have some common identifiers in them. In bound expressions there was absolutely no reason why the same identifier could not have been bound differently in different localities in different sub expressions. Let us assume that (row) P_1 is an environment over v_1 which is represented as $P_1 : V_1$. Row 2 is an environment over V_2 which is represented as $P_2 : V_2$. Then we will define operation called row 1 with row 2 within brackets $P_1[P_2]$. We will define $P_1[P_2]$ as an operation.

I have taken out the minimum information that I require. As far as I am concerned, an environment is not a data structure but is an abstract entity. After you have semantically

specified all that you require of an environment you can come out with a suitable data structure to represent environments. I will define P_1 , P_2 and $P = P_1[P_2]$ as an operation on environments which creates a new environment. For each identifier 'i' if $i \in V_2$ the binding given by a row is the same as the binding given by row 2. For all identifiers that occur only in V_1 and not in V_2 , the binding given by a row $P(i)$ is the same as the binding given by $P_1(i)$. We are only considering the set of identifiers in $V_1 \cup V_2$. (Note $P_1[P_2]$ is an asymmetric operation which is not the same as $P_2[P_1]$).

(Refer Slide Time: 39:43)



If you look at our grammar, expressions require declarations and declarations also require expressions so we cannot isolate the two completely. Any semantics of the expression language will depend upon the semantics of the declaration language and any semantics of the declaration language will depend upon the semantics of the expression language.

Now we have to give a holistic semantics including both expressions and declarations and we have to make it complete. In exactly the form we have a grammar, the main language is a language of expressions and I will start with the transition system for expressions. Now I have a difference. I have to consider only all those expressions in the language over some set of variables v . Very often I will just get rid of v .

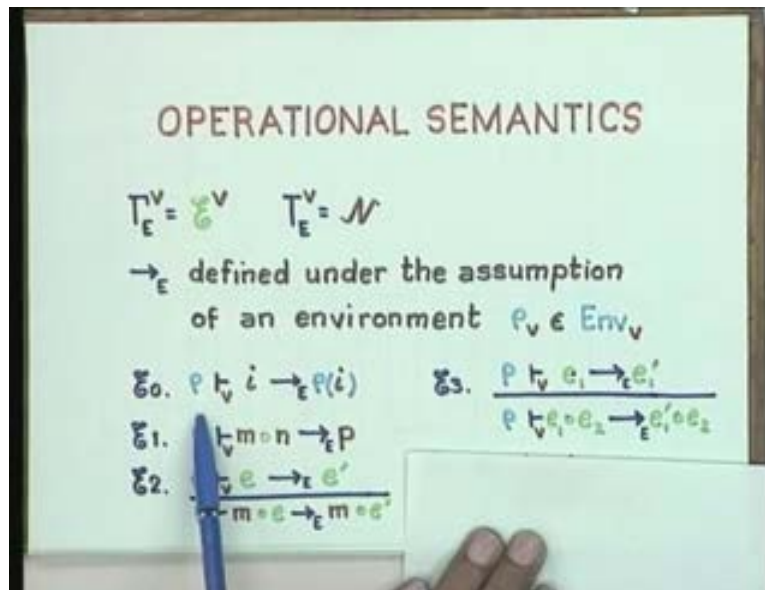
The set of all configurations of expressions is just the set of all expressions over v and the set of all terminal configurations is still the set of values because the meaning of an expression ultimately is just a value. We will define a transition relation under the assumption of an environment. An expression which contains identifiers does not have a meaning unless each of those identifiers has a meaning. In other words the expression does not have a value unless each of the identifiers has been given a value and that value assignment to the identifiers is going to be in the environment row.

I will just quickly go through the modified forms of the transition system. First we have to introduce an axiom which says that in any environment row where an environment is just value binding to each of the identifiers, the set of value bindings, an expression which is just an identifier allowed by our grammar has a value just given by the binding. Note I said before that our transition systems are syntactic or symbol manipulations and whenever you encounter the identifier the transition indicates that that is really being replaced by the value given by the environment.

It is still symbol manipulation but symbol manipulation in the abstract sense. The other three rules are exactly as we had before, except that now we have to bring in the environment because the rule $e1$ does not really require an environment. However, an application of $e1$ often in the general case would never be complete without an environment because $e1$ would be applied only after you have successively applied $e3$ and $e2$ several times. Just go back to our basic example of our transition system. We are specifying a left to right method of evaluation. So, given an expression of the form $e1$ a binary operation, $e2$, you would keep applying $e3$ till the left hand operand became a constant and became a numeral. Then you would start evaluating the right hand operand till that became a constant and then you would just evaluate the constant. So, there would be several applications of $e3$, till the left hand operand became a constant then several applications of $e2$, till the right hand operand also became a constant and then an application of $e1$. That is what would happen in the syntactic application of these rules in a typical large expression. In the case of the introduction of identifiers before an

expression like e_1 became a constant there would have to be an application of a call to an environment somewhere to replace the identifiers in it by appropriate constants or literals.

(Refer Slide Time: 42:32)



There would be interspersed in the process of execution applications of e_0 where identifiers are replaced by constants and all of them require the environment row. However, this is only for the basic language of expressions which does not contain any declarations. We have to still give a meaning for the let construct. However, the value of an expression depends upon the declaration inside the let construct and so there is no way I can evaluate the expression till I have evaluated the declaration and evaluating the declaration means creating fresh bindings. This means that the expression can make some progress only if the processing of the declaration can make some progress.

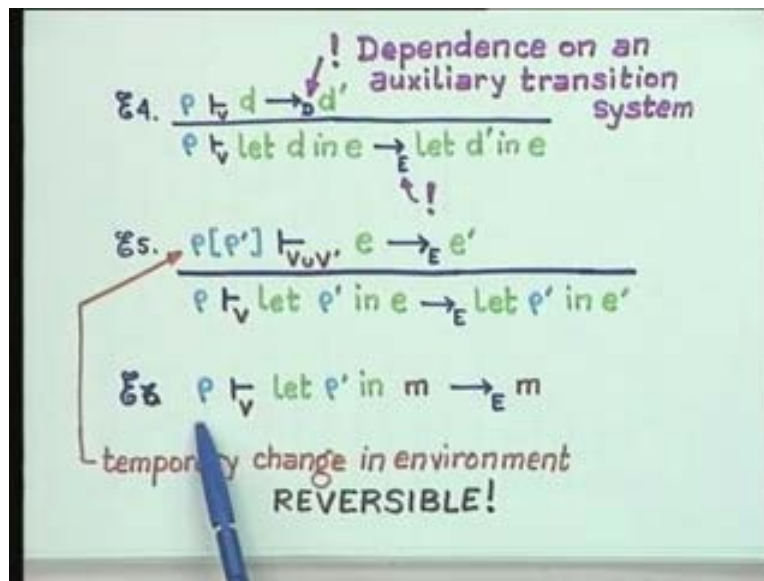
I have not yet specified what it means for a declaration to move. Supposing several applications of e_4 will finally reduce the declaration to some environment row prime then I am in a position to evaluate the expression and it can be moved to row prime. I evaluate the expression really in the environment row updated with row prime. However, the entire expression, 'the let construct' really is executed in the original environment row with which it came. In the case of a completely closed program the row would be an

empty set. But in the case of a program which is not completely closed, for an expression need not be completely closed, then it would execute in some environment row.

Once the declaration has been completely evaluated to produce an environment row prime the expression can be evaluated and the evaluation proceeds in the updated environment row prime and in a modified environment essentially if there are no let constructs within the expression and there are no nested bindings then you would be applying rules to finally evaluate the expression by several applications and having done that you will be left with a constant finally and the result of the expression is a constant. But note that its environment is only row and the updation row prime is really a temporary change in the environment and in that sense it is reversible. So, this temporary change in the environment is necessary in order to evaluate the expression but the environment in which the entire let construct is being executed is in the original environment that it started out with.

The updation of environments is a typical case of a reversible change in the sense that if you apply these rules systematically there is absolutely no reason why you should be left with a strange updated environment which is different from what you started of originally with. You always end with the original environment you started out with.

(Refer Slide Time: 50:25)



In a completely closed PASCAL program you start with an empty environment and after you have executed you still have nothing more than the empty environment. The completely closed PASCAL program executes in the original global environment and as declarations come new updations, which are temporary, are created and as those scopes are exited those updations go away automatically. I will continue on the 'declaration transition system' in the next lecture and we will look at some examples.