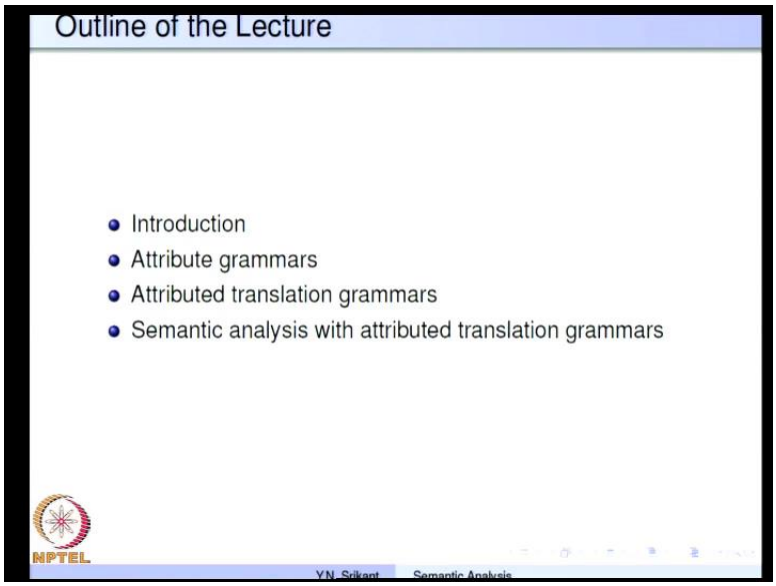
Principles of Compiler Design
Prof. Y.N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

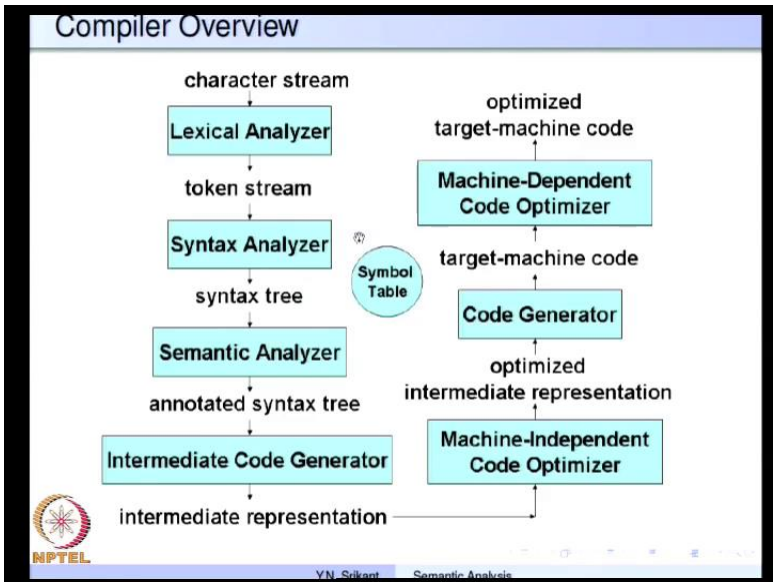
Lecture - 12
Semantic Analysis with Attribute Grammars Part-1

(Refer Slide Time: 00:21)



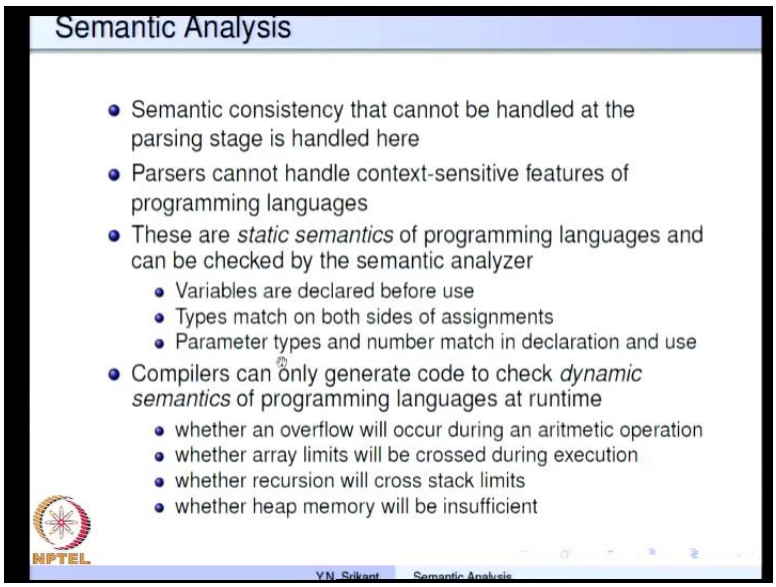
Welcome to the lecture on semantic analysis. We will be studying many topics such as attribute grammars, attributed translation grammars, and how exactly semantic analysis is done with such attributed translation grammars. These are the main topics and as we go along, we will see the sub topic as well.

(Refer Slide Time: 00:39)



To put the semantic analysis module in the correct prospective, let us look at the overview diagram once again. So, we have completed lexical analyzer, syntax analyzer, and studied their properties different type of such analyzers, and so on. So, one from this syntax analyzer, the syntax tree is input into the semantic analyzer module, which outputs these annotated syntax tree after validating the semantics of the program. So, let us understand what exactly semantics of a program mean? How they are checked, etcetera, etcetera.

(Refer Slide Time: 01:22)



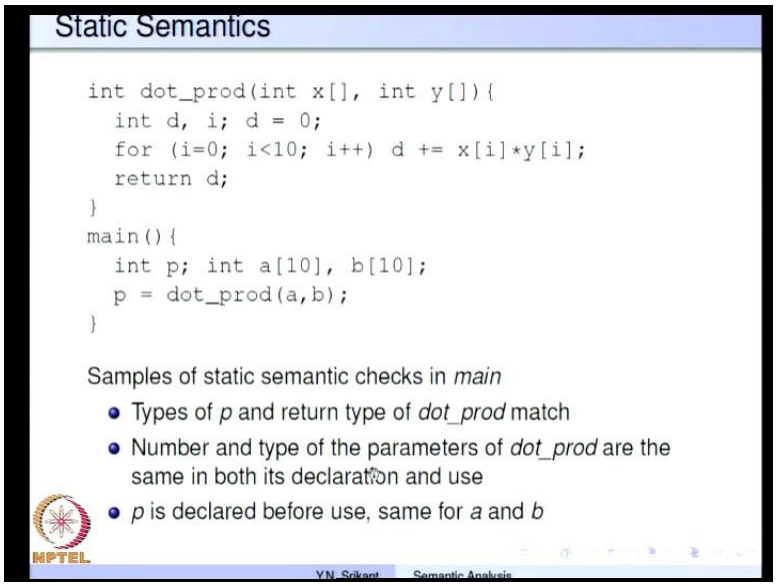
So, basically semantic consistency that cannot be handled at the parsing stage is actually the one that is handled here. For example, parsers cannot handle context sensitive features of programming languages. So, there are two types of semantics that programming languages have; one is known as the static semantics and other is known as the dynamic semantics. Static semantics as the name indicates you know they do not depend on the run time system and the execution of the program, but they are depended only on the programming language definition. Whereas, the dynamic semantics again as the name indicate they are the properties of the programming language systems that occur at run times, and we need to check such properties only during execution time of the program.

For example, the static semantics can be checked by semantic analyzer or you know there are many examples here. So, all variables are declared before use, if so everything is otherwise an error message has to be provided. Then do the types of the expression and the variable to which it is assigned match on the two sides of an assignment statement and do the parameter types, and number of the parameters match in both the declaration and use.

So, these are few examples of static semantics of programming languages where as you know for dynamics semantics compiler can only generate code to check such you know meaning of the program. For example, whether an overflow will occur during an arithmetic operation, so whether array limits will be crossed during execution, whether recursion will cross stack limits, whether heap memory will be sufficient.

So, really speaking we should say you know the properties are these and the checks will be done by the semantic analyzer and similarly you know there are properties of the numbers and if the number exceeds a particular range, than over flow is set to occur. Similarly, the array will be defined which says 0 to 10 or 0 to 20 and if the program really crosses these limits, then the array limit violation is set to occur. So, that the violation should not happen is the semantics of the program, where as that check will performed only at the run time.

(Refer Slide Time: 04:27)



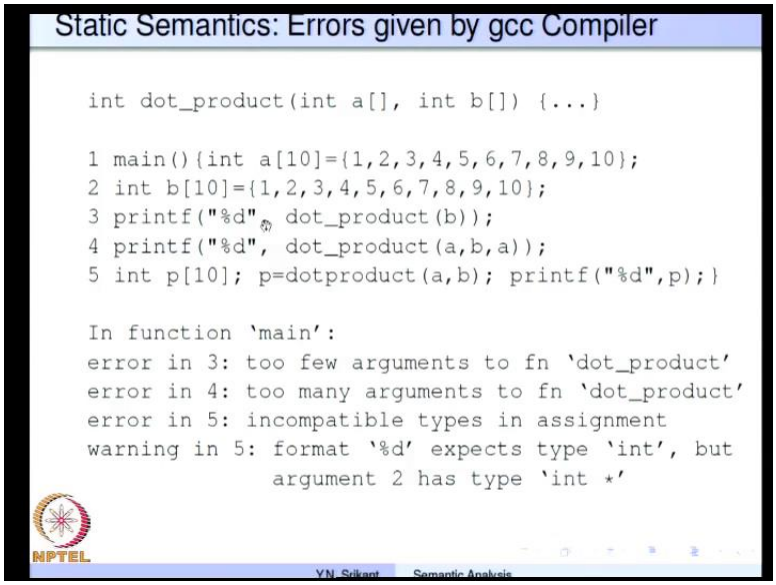
So, let us take some examples of the static semantic checks that can be done by a complier. So, consider this you know simple function and main program which calls it. So, this is a function to calculate the dot product of two arrays, so int X and int y this are the two arrays of integers. So, where the program is very simple you know it has a loop i equal to 0, i less than 10, i plus plus and then d accumulates the dot product d equal to d plus X i star y i, so return d.

So, this is the simple you know dot product function and in the main program we have two arrays of a and b of size 10 each and we call dot product function with this two arrays as parameter the sum is actually taken in p. So, what are the possible static semantic checks that can now performed on this program. For the main program here is a simple you know list, so of types of p and return types of the dot product function they should match so.

For example p is being assigned the value of a value return by a dot product and p value is integer. So, the return value of dot product is also of integer type, so these two matches and there is no question of error here. Do that numbers and type of the parameters of dot product the same in both its declaration and use, yes in the case, so. So, here is the usage, in this usage we have two parameters and in the declaration of dot product also we have two parameters.

Secondly, the 2 parameters are arrays in the declarations and they are arrays in the usage as well, so this particular properties also satisfied. Then is p declared before use? Similarly is a declared before use? Is b declared before use, etcetera. So, yes definitely p is declared and then used, a and b are declared and then used. So, all this properties are satisfied for the main program.

(Refer Slide Time: 07:02)



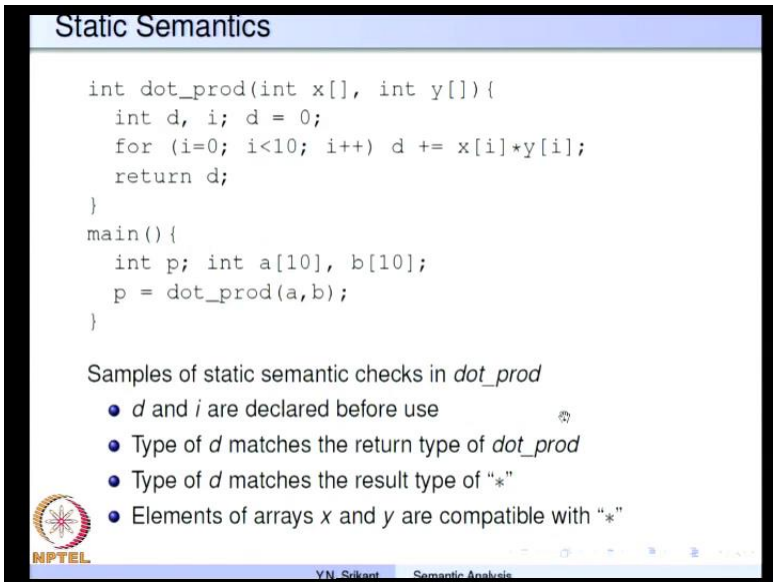
So, let us see how it is further erroneous program. So, the same dot product let us not worry about the dot product function, so I have just you know given you only the declaration of the header. In the main program there are many errors here and the error messages given out the g c c compiler are here. So, for example, we have a and b as before and then we are trying to call dot product with a single parameter, here we are trying to call at with three parameters, here we are calling it with you know with two parameters properly. But unfortunately the p into which we are trying to accumulate the dot product is not a single integer, but is an array type.

So, let us see and of course print f says print an integer and p and array is provided as a parameter for the print f statement. Let us look at the errors which are given out. So, statement 3, too few argument to function dot product, so its require two, but we have given only one, so the semantic analyzer in the compiler has got the error.

Then statement 4, it says too many arguments to function dot product it requires two, but we have provided three, again an error and statement 5, incompatible types in

assignment, so p equal to dot product, p is an array where as dot product written as an integer, so the types do not match on both sides. So, that is the error which being printed out here and this is the warning really for the fifth one, for the print f statement. So, format percentage d expects type int, but argument 2 has type int star. So, P is an array, so it is consider as type int star and that is the reason why it says has type int star. So, these are the examples of errors in the main program.

(Refer Slide Time: 09:14)

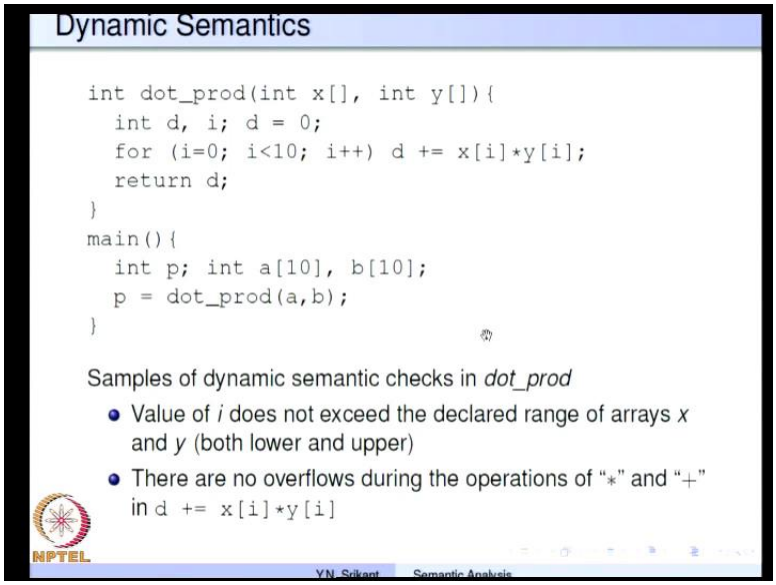


Continuing the same program in the function dot product, again there are many semantics checks possible even though it is a repetition, let just go through it to drive home the point. So, are d and i declare before use, so d is being initialized here, sorry d is being declare here and assigned here. So, declaration happens before usage i is declared here and then used here, so its property of declaration before use is satisfied again. The same is true for even you know the X and Y, because X and Y happened to be appearing as parameters. So, they it is in some sense a declaration of these two variables.

Then type of d matches the return type of dot product, so again d is being returned by the function, so the return value is type int for the function that product and what is being returned is an integer again int, so this property also is satisfied. Then type of d matches you know a result type of star as well. So, for example, here we say d equal to d plus X i star Y i. So, that means, we are multiplying X i and Y i, adding it to d and then you know the result is again, the value is stored in, updated again and again in the far loop. So, star

is operating on two integers and it produces an integer and then it is added to d. So, int plus int is again a valid combination, therefore this property is also holds. Elements of arrays X and Y are compatible with star, so again this is a proper match, because X i and Y i are integers and the star between two integers is valid. So, these are some example of semantics checks static semantic checks in the functions dot product.

(Refer Slide Time: 11:23)

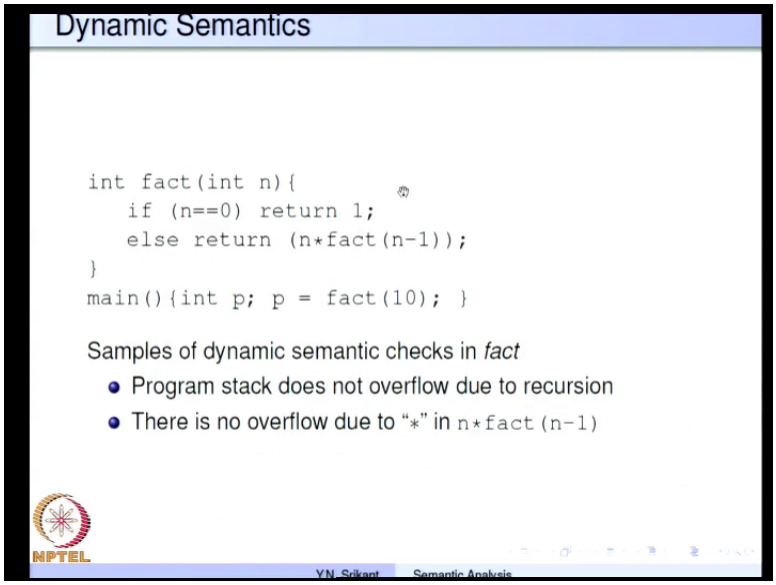


So, let us you know consider some of the dynamic semantic checks in dot product. So, value of i does not exceed the declared range of arrays X and Y both lower and upper, so X and Y in the main program have been declared with size 10. And for some reason if we say this is the size 11 or 12, i less than 11 or i less than 12, then you know we would be accessing X 11 and Y 11, X 12 and Y 12, etcetera, which are not within in the range of the declarations.

So, there would be a run time error at this point, in this case of course, X you know i goes from 0 to 9 and that is only 10 iterations. So, everything is well within the you know declaration of the two arrays X and Y. There are no overflows during the operations of star and plus in d equal to d plus X i star Y i. Again the integers that we are, if the values of the variables a and b, the arrays of components of a and b happened to be extremely large values. When the multiplication X i star Y i could actually cause an overflow to occur.

Similarly, again if these values, if the array are too large and to many components are being multiplied and added into d, then d could again you know cross the limit and cause an overflow, but in this case assuming that a and b have small values such overflows will not occur. So, these are some example of what exactly are dynamic semantic checks in programs that can be, that cannot be done by the compiler, but the compilers definitely generate code to perform these. For example, what happens is to check the array going to out of bounce at run time the index value of i is checked to be less than 10, if it is then the iteration is allowed to go further otherwise an error is caused and the program stops.

(Refer Slide Time: 13:57)

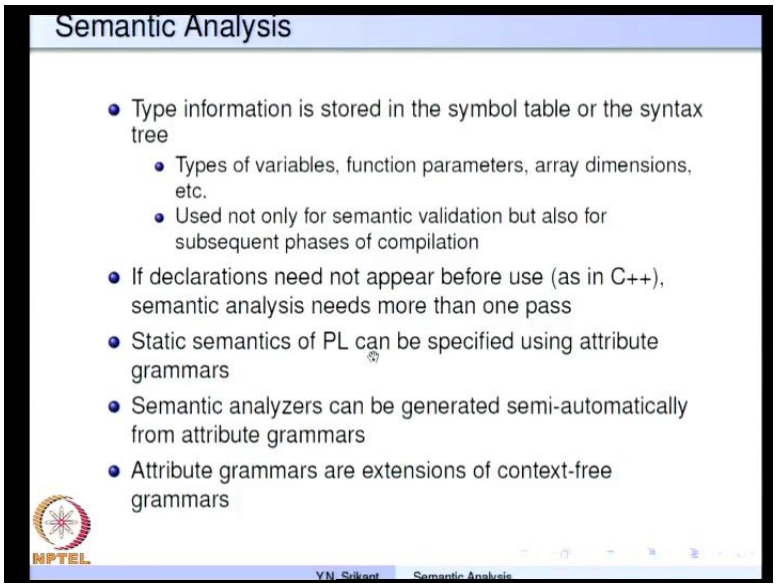


So, another type of dynamic semantic check that can take place is again demonstrated with the use of recursive function int fact. So, this compute factorial of a number, so if n equal to 0 return 1, otherwise return n star fact of n minus 1, this is the very well known program, so there is a no need to explain further. So, p is an integer and p is equal to fact 10, so we are trying to compute 10 factorial. So, thus the program stack overflow due to recursion, so here, so when we say a fact of 10 it comes here 10 into fact of 9, then again 9 into fact of 8, etcetera, so there are really 10 invocation of fact. So, therefore, will the stack permit 10 instances of the activation records, which are required to execute the function fact that is the question?

So, if does not then there will be an overflow otherwise the program will go through 10 is a small number, so there is no need to worry about overflow in this particular case,

there is no overflow due to star in n star fact n minus 1. So, again if n into fact of n minus 1 goes beyond the range of represent able integers, then there would be an overflow at this point.

(Refer Slide Time: 15:31)



So, that is an introduction to what exactly are semantic checks. So, let see what exactly semantic analysis. So, during semantic analysis the type information of the variables whether there are int or arrays or Booleans or characters, etcetera is stored in what is known as the symbol table or the syntax tree.

So, it can be stored in the symbol table or it could be stored in the notes of the syntax tree itself, usually the symbol table option is better. Because syntax tree takes much more space then the symbol table, usually we store it in a symbol table and then place a point to the appropriate in to the symbol table in one of the fields in the nodes of the syntax tree. So, what is information that is stored? Types of variables, function parameters, array dimensions, of course the variable name etcetera are also stored in the symbol table.

So, these symbol tables are used not only for the semantic validation or semantic analysis, but there are also required and used for subsequent phases of the compilation. For example, during intermediate code we will defiantly access the symbol table to check and you know get the information regarding the size the of arrays, the dimension, number of dimension of the array, etcetera, type of the element of the array. All this are

required for code generation, even intermediate code generation and that would be our purpose to access the symbol table during intermediate code generation.

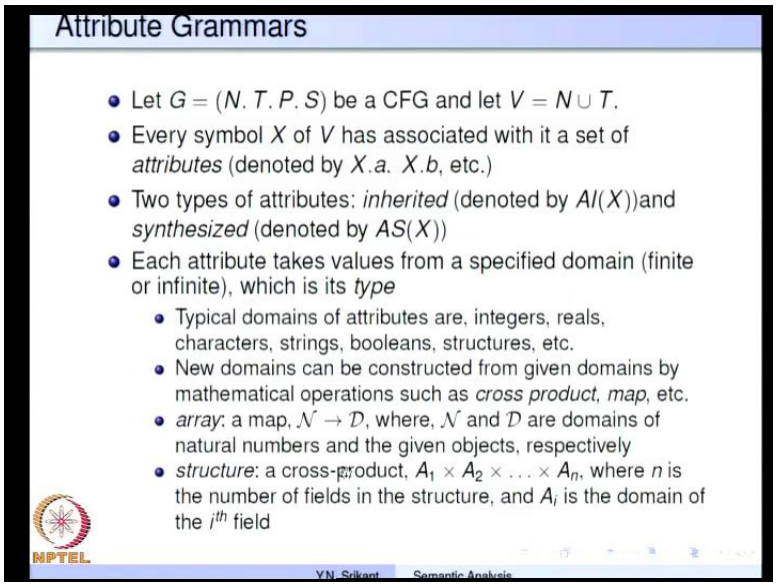
During machine code generation again we will need to know the types of the variables in order to allocate space for them, etcetera. So, optimization again requires the information about variables for moment of code, etcetera. So, all these are uses of the symbol table during other phases of compilations. Then if the declarations you know need not appear before use as in the case of the language C plus plus, then semantic analysis requires more than one pass.

The point is when you write class declarations in C plus plus, you put the member functions and then you know usually you write all the declarations of the variables that are used within the methods. So, in such a case when we try to pass the methods or member functions we will not know the type of variables, because the declaration appears latter. The only way to handle it is to go through the program once create the symbol table and then go through the program second time and perform semantic analysis.

So, we will see more of this toward the end of these lecture has so static semantic of programming languages can be specified using what are known as attribute grammars. So, we will see how to check the static semantics of programming languages and how to specify them using attribute grammars.

Semantic analysis can be generated semi automatically from attribute grammars. So, we write the semantic you know specifications for the program for the programming language and then we could use a generator which generates them from such an attribute grammar. The reason we say semi automatically is that the semantic are usually specified in a programming language like notation it is not a pure side effect free notation, so that is the reason why it is called semi automatic. Programmer has to sequence the semantic checks appropriately and then submit to the generator. In fact, yacc also is a semantic analyzer generator if is supply the semantics checks appropriately in C. Attribute grammars are extensions of context free grammars. So, now let us move on and study attribute grammars in some detail.

(Refer Slide Time: 20:16)



So, happens that attribute grammars require some terminology to be understood before we pick up an example and study it. So, let us go through some terminology understand it and then proceed with an example. So, let G equal to N T P S be a context free grammar, as I said attribute grammars are nothing but extensions of context free grammars. So, the base is definitely a context free grammar and let the set of variables V of the grammar be N union T. For every symbol of for X of V we can associate a set o f attributes denoted as X dot a, X dot b, etcetera, that is why the name attribute grammar. There are two types of attributes inherited attributes which are denoted as AI of X, so inherited attributes of X and synthesized attributes which are denoted by AS of X, these are really sets of attributes.

Each attribute takes values from a specified domain, it could be a finite domain or it could be an infinite domain and we call the domain you know such a domain as its type. For example, even in programming languages we say int of x, so x integer domain is the domain from which the value of x, we provide a values for x in our program. So, we could say integer is the type of the variable X, the same notion is carried over to attributes to as well. So, typical domains of attributes are integers, reals, characters, strings, Booleans, structures, etcetera.

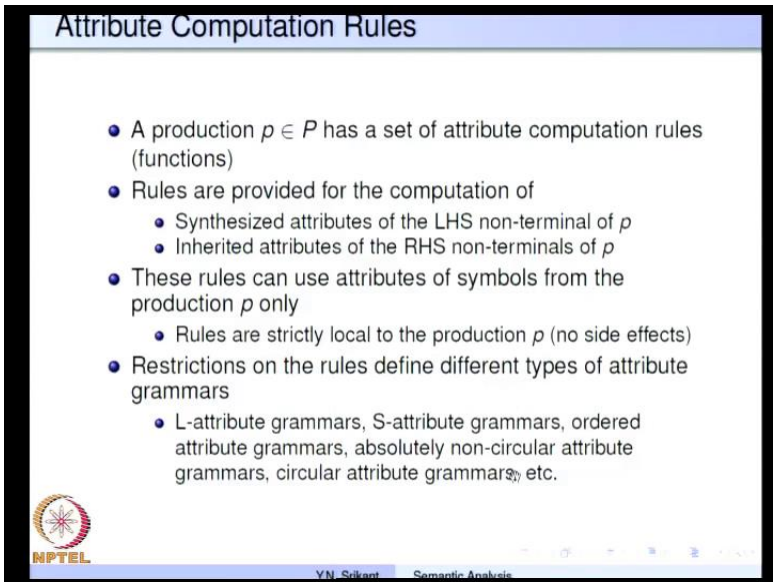
Now, given some basic domains such as integers, reals, characters and booleans we could actually construct new domains from the given domains using mathematical

operations, such as cross product, map, etcetera, so even union, intersection, all this are permitted on these domains. There are two examples which I provide here one is for an array, the other is for a structure. So, you can think of a single dimensional array as a map form this domain of a natural numbers to the domain of our objects, so N to D.

So, basically for every natural number we provide the object which is to be placed in the array location. So, if we say a is an array a of 1 would contain some object that we want, a 2 will contain some other object we want and so on, where N and D are domains of natural numbers and the given object respectively. So, we could placed restrictions on the number of elements in N define a finite subset of N here and then similarly this will also be a finite set, so in that case we would have a finite map. If we want two dimensional arrays, then instead of a single domain here you would probably have a cross product of domains i cross j cross k, you know that would mean i comma j comma k that would be from N cross N cross N to D.

So, you take two or three dimensions, than appropriately the cross product is also provided here. If you take a structure then it can be obtained using a cross product operation, just like an array was obtained using a map operation. So, the cross product is A 1 cross A 2 cross etcetera a n, where n is number of the fields in the structure and A i is the domain of the i'th field. So, if you have a structure with two integers and one character then A 1 would be integer, A 2 would be integer and A 3 would be char. So, this would again be finite objects or infinite, you know finite domains or infinite domains as the example requires.

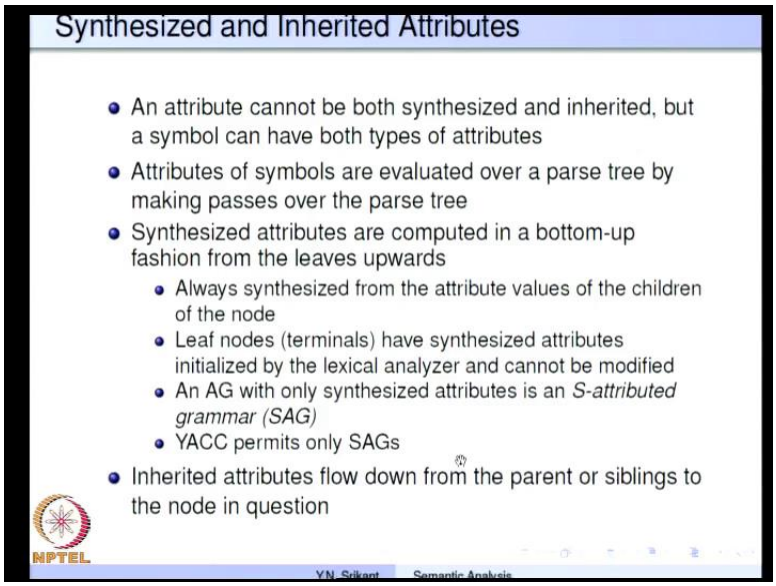
(Refer Slide Time: 24:54)



Now, production p has a set of attribute computation rules or functions. So, this is a basically the rule which is provided to compute a value for appropriate attributes of the production. So, rules are provided for the computation of synthesized attributes of the left hand side non terminal of p and inherited attributes of the right hand side non terminals of p. So, for example, you cannot compute the inherited attributes of the LHS non terminal, similarly you cannot compute the synthesized attributes of the RHS non terminals of the production p, so this will become clear as we go along. These rules can use attributes of symbols from the production p only. So, in other words the rules are strictly local to the production p, there are no other side effects as well.

So, you cannot access attributes of others symbols from different productions, you can access the attributes of various symbol within the production. Restrictions on the rules really define different types of attribute grammars. For example, we are going to study the L-attributed grammars we are also going to study S attribute grammars, but we will not worry about the other types. There are ordered attribute grammars, absolutely noncircular attribute grammars, circular attribute grammars, etcetera ,there must be more than half a dozen such variety is possible.

(Refer Slide Time: 26:41)



So, now let us understand what exactly synthesized attributes are and inherited attributes. So, an attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes. So, if x is a non terminal it can have a as a synthesize attribute and b as inherited attribute, but a single attribute cannot be of both types. Attributes of symbols are evaluated over a parse tree by making passes over the parse tree; this is a very important point to note. So, in general the parse tree is always assumed to be available from the parsing stage, then the various nodes in the parse tree, internal nodes are all non terminals and the leaves are all terminals symbols.

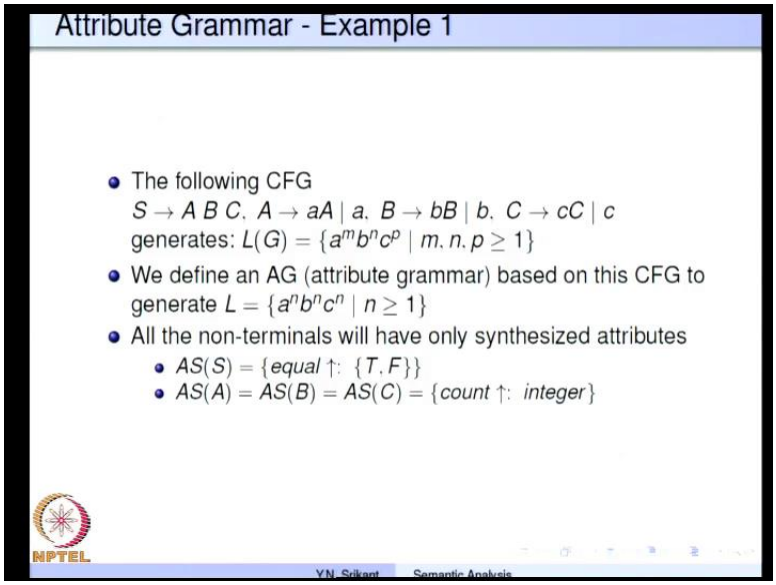
So, they will be associated with many attributes, so these attributes must carry you know must get appropriate values through computations. So, that means we must also provide an order in which these attributes will be evaluated over the parse tree.

Synthesized attributes are computed in a bottom up fashion from the leaves upwards that is why they are called synthesized, so always synthesized from the attribute values of the children of the node. Leaf nodes which are terminals symbols have only synthesized attributes, they really cannot have inherited attributes and these you know synthesized attributes are initialized by the lexical analyzer and cannot be modified.

So, for example, if you take a name, the character string associated with that name is returned by the lexical analyzer and the token corresponding to the name say identifier, would have synthesized attribute, which is the character string for that particular name.

Similarly, if you take an integer constant then the token const, int const would have an attribute which is the value of that particular constant and again this value is returned by the lexical analyzer. An attribute grammar with only synthesized attributes is an S attributed grammar; this is a very important class of attribute grammars. It so happens that the YACC tool permits specification of only S attributed grammars; it does not permit specification of inherited attributes at all. Inherited attributes on the other hand flow down from the parent or siblings to the node in question. So, let us study the synthesized attributes first and then move on to the inherited attributes.

(Refer Slide Time: 29:45)



So, let us take an example to understand how these synthesized attributes and in general how an attribute grammar is specified. Take a context free grammar S going to A B C, A going to a A or a, B going to b B or b, C going to c C or c, this generates the language a to the power m, b to the power n, c to the power p, where m n and p are greater than or equal to 1 and observe that there is no relationship between m n and p each one of them varies depending on these string.

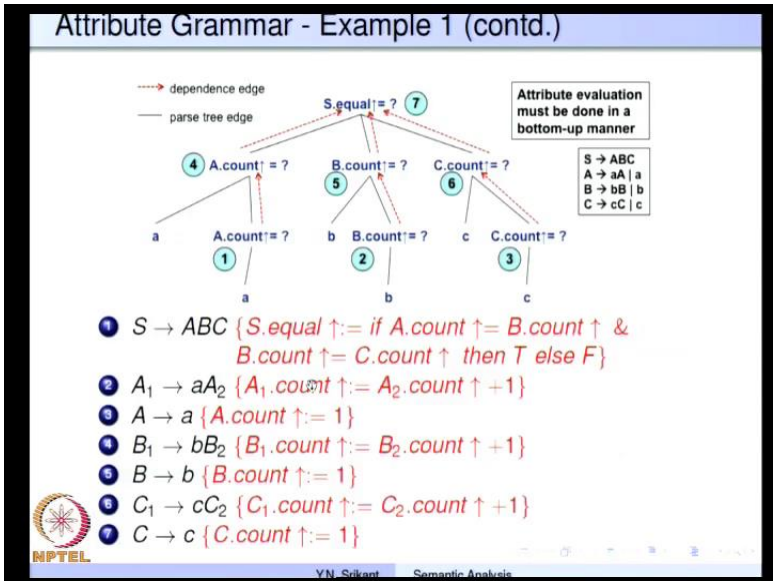
Suppose, we want to generate a to the power n, b to the power n, c to the power n, that is the counts of a, b and c are equal and n greater than or equal to 1. Let us define this is obviously known to be a context sensitive language; it is definitely not context free. So, no matter what modifications we make to this context free grammar to get another context free grammar we will never succeed. So, no context free grammar can ever

generate a n, b n, c n, but it so happens that an attribute grammar based on this context free grammar can be defined, so that the language a n, b n, c n is generated. So, for this grammar the synthesized attributes are the only once we require. So, the synthesized attributes of S just 1 the equal, the attribute name is equal, the arrow here indicates, up arrow indicates that this is a synthesized attribute, similarly a down arrow would indicate it that it would be an inherited attribute.

The domain of values for this attribute is T comma F true or false, a finite domain; the non terminals A, B and C again have just 1 attribute count. So, A has an attribute count, B also has an attribute count, C also has an attribute count, it does not mean they share the attribute, they are different attributes, but the name is the same, that is perfectly, ok. Any terminal or non terminal all of them can have attributes with the same name, but it does not mean that they are shared.

Count is a synthesized attribute and the domain of values is that of the integers, so any integer value is possible for count. So, typically when we generate the basic idea is this, when we generate A using the rule a going to little a or A. We associate attribute computation rules such that the number of A is counted, the same is true for B and C and when we come to the root, we check whether the number of A equal to number of B equal to number of C, if so then the string is accepted, otherwise the string is rejected.

(Refer Slide Time: 33:17)



So, here is the attribute grammar, this also shows how attribute computation rules are positioned within the grammar, so we will come to this diagram after we understand the attribute grammar. The rule says S going to A B C, the context free grammar rule, so now within these flower brackets we write the attribute computation rule. So, the remember that the synthesized attributes of the left hand side non terminal need to be provide with attribute computation rules, but we do not have any inherited attributes on the right hand side we have only synthesized attributes. So, there is no need to provide any rule for the computation of attributes of A B and C.

So, just one rule here, S dot equal that is the notation we use to indicate the attribute of S, the up arrow indicates that it is a synthesized attribute. So, this is computed as, if A dot count equal to B dot count and B dot count equal to C dot count, then true otherwise false. So, the rule is very clear if A B and C, A dot count, B dot count, C dot count are all equal, then S dot equal becomes true otherwise it becomes false. So, if S dot equal is true, then the string is in the language and if S dot true is false then the string is not in the language.

For the production A going to a we say A dot count equal to 1, because this is generating exactly 1 a. For the production A going to a A we are differentiating the two occurrences of the capital A, we provide a rule for the computation of A 1 dot count, we do not provide any rule for the computation of A 2 dot count, because that is a synthesized attribute on the right hand side. So, A 1 dot count is whatever is the number of A already generated by this A 2 plus 1, so A 2 dot count plus 1. Similarly, the rule for the number of B is B 1 dot count equal to B 2 dot count plus 1 and for the C, it is C 1 dot count equal to C 2 dot count plus 1. Now, let us look at the parse tree and understand how the attributes are positioned on the nodes of the parse tree.

So, if the here are the leaves, so there are 2 a, 2 b and 2 c, so the sentence is indeed in the language. So, here this is A dot count, because this is the non terminal A and here this is again another non terminal A, so this has the attribute A dot count, etcetera. So, similarly B dot count S, C dot count S, S dot equal, etcetera. To begin with the attribute instances do not have any value, because we have not carried out any attribute evaluation, the attribute dependences are all shown in red. So, for example, the A dot count in this particular node requires the value of A dot count at its child node, so this is very clear from the production that is used. So, the production used here is A going to a A, so if you

look at the a production A going to a A, the attribute computation rule says A 1 dot count equal to A 2 dot count plus 1.

So, this A 2 is nothing but the child node one of the children of the particular node A. So, this has two children a and this A, so this count is required to compute this count. Whereas, the production used at this point is just A going to a, so which has just A dot count equal to 1, so there is no need to use any other attribute value to compute this A dot count. The same holds for this B dot count, this B dot count, this C dot count and this C dot count. Finally, S dot equal requires the computed values of the three counts here, that is the three children A dot count, B dot count and C dot count. So, there are three dependences shown for S dot equal.

So, let us now see how to evaluate these attributes. So, obviously, because this attribute requires this attribute, it is necessary to evaluate this attribute before we go the evaluation of this attribute, the dependence indicates that. So, this attribute can be evaluated very simply as A dot count equal to 1, similarly the b attribute can be evaluated as B dot count equal to 1 and this attribute can be evaluated as C dot count equal to 1. So, the three attributes 1, 2 and 3 can be evaluated you know in three steps in any particular order, but they must be evaluated first before we go the next level. So, this is what we mean by a bottom up evaluation.

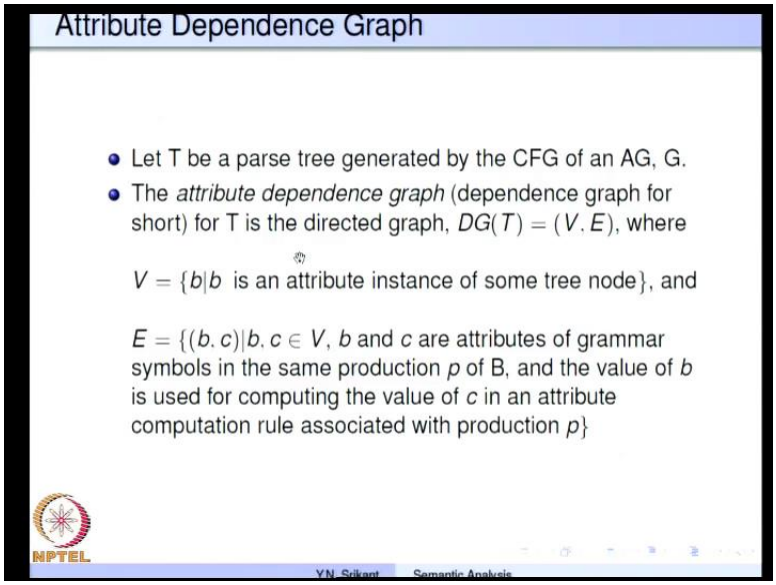
In the step number two, so this was already computed, we compute these attributes, so this A dot count is this A dot count plus 1, so this becomes 2, similarly this also becomes 2 and this becomes 2 as well. So, we can compute 4, 5 and 6 these attributes in any particular order, you can compute 4 first, then 5, then 6 or you could compute 4, then 6, then 5 or you could do 5, 6, 4, it does not matter, but we must complete the evaluation of the children before we go to this particular level.

Once we have completed the evaluation at this level, now we are ready to compute the, you know attribute value at the root, so these have been completed. The S dot equal rule says if A dot count, B dot count and C dot count are equal, then true else false, in this case all these three are equal, so it is true.

So, what we must observe here is the local dependence of the attributes in the attribute evaluation rule. So, for example, to compute this we require the counts only at the next immediate level, we will never try to accesses the attributes at two levels lower. So, to

compute this we require this attribute which is one level lower, etcetera. So, this is what we mean by the locality of the attribute computation rule.

(Refer Slide Time: 40:41)

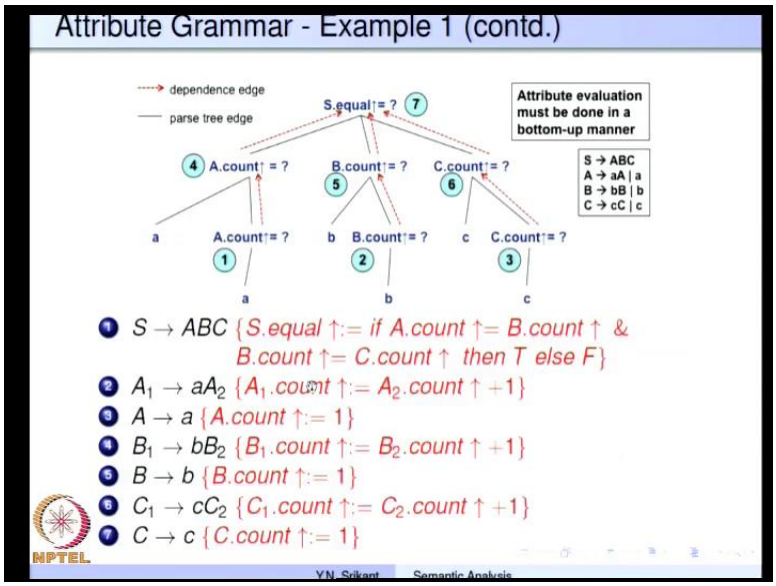


So, let us move on and now we are going to look at the attribute computation in general. So, let T be a parse tree generated by the context free grammar of an attribute grammar G, so now you know lets go back for 1 second.

So, I am sure you realize that the attributes can be located on the parse tree and only then can they really be evaluated. So, we are assuming that the parse tree is computed by the syntax analyzer and passed on to the semantic analyzer and then onwards we take over and compute the attributes over the parse tree. So, let T be a parse tree generated by the C F G of an A G, G.

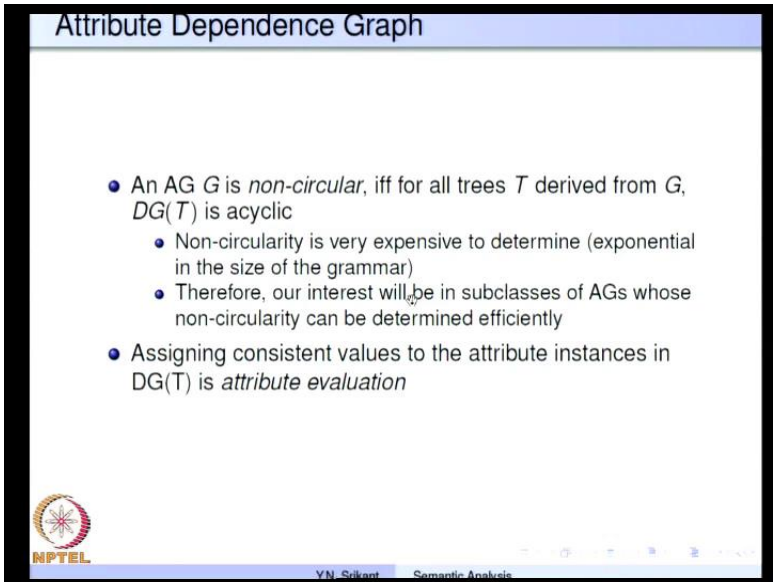
Now, we must define the attribute dependence graph or dependence graph for short for T. So, we have shown it informally in the previous example, so the D G, T is a graph V comma E where V is the set of b, such that b is an attribute instance of some tree node. So, all the attribute instances of the nodes happen to be, you know the nodes of the vertices of this particular direct graph. The set of edges is b comma c, where b and c are in V and b and c are attributes of the grammar symbols in the same production p of B and the value of b is required for the used for the computing the value of c in an attribute computation rule associated with production p.

(Refer Slide Time: 42:42)



So, if we go back to this example, these red ones are all the dependence graph edges. So, 3 plus another 3 here, so these 6 edges actually belong to the dependence graph. So, the attribute instances are all labeled with numbers here, so those are the vertices. So, remember even though the parse tree has other terminals a, b and c these are really not part of the attribute dependence graph, because they are not required in the computation of the attributes. In case these terminal symbols carry attributes, which are required for the computation of some attributes, then those attributes of the terminal symbols would also be a part of the dependence graph.

(Refer Slide Time: 43:31)

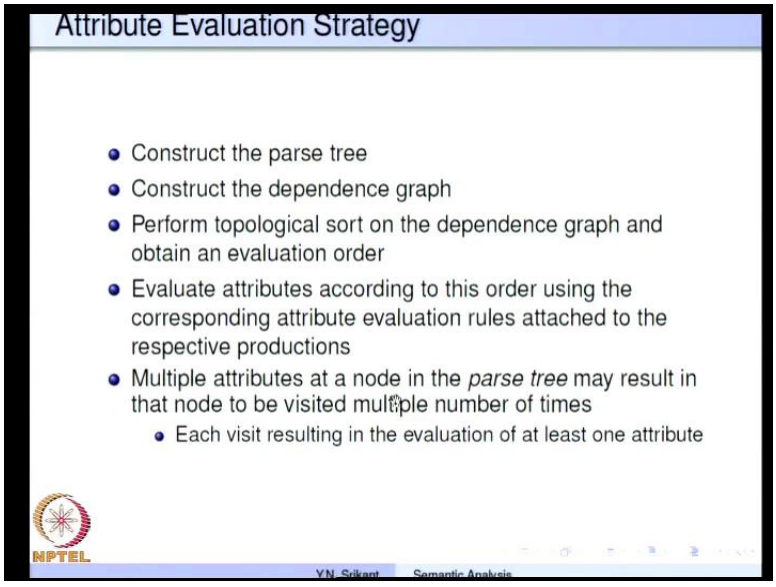


So, that is the dependence graph, but what about it? An A G, G is called as a non circular attribute grammar, if and only if, for all trees T derived from the grammar G the dependence graph T of T is acyclic. So, this is quiet you know easy to perceive, because if the dependence graph is cyclic, then we may not be able to find an order in which the attributes have to be computed.

So, just assume that S dot equal is required for the computation of A dot count as well at this level, so then there would be a edge here in the downward direction and we would have a cycle. So, that means the attributes on this cycle cannot be evaluated, that is the reason why we place this property of circularity. So, non circular attribute grammar are the only ones which can be used in practice, but still even checking for non circularity is a very expensive algorithm, so they are exponential in the size of the grammar.

Therefore, our interest will be in the sub classes of attribute grammars whose non circularity can be determined very efficiently, so this will become as we clear as we go along. Assigning consistent values to the attribute instances in the dependence graph of T is called as attribute evaluation, we have informally seen what is attribute evaluation.

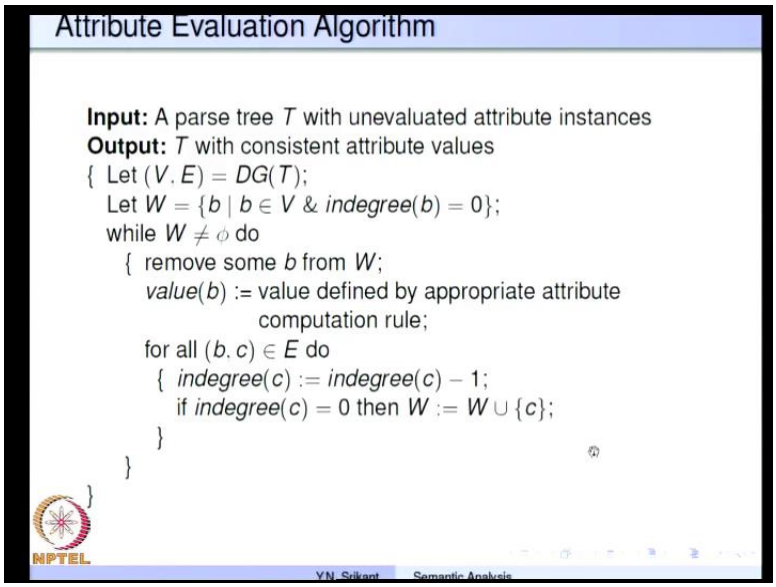
(Refer Slide Time: 45:12)



So, let us look at attribute evaluation algorithm in general. The process is simple, construct the parse tree, then construct the dependence graph, perform topological sort on the dependence graph and obtain an evaluation order. Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective

productions. Multiple attributes at a node in a parse tree may result in that node of the parse tree to be visited multiple number of times, but each visit resulting in an evaluation of at least one attribute of that particular node.

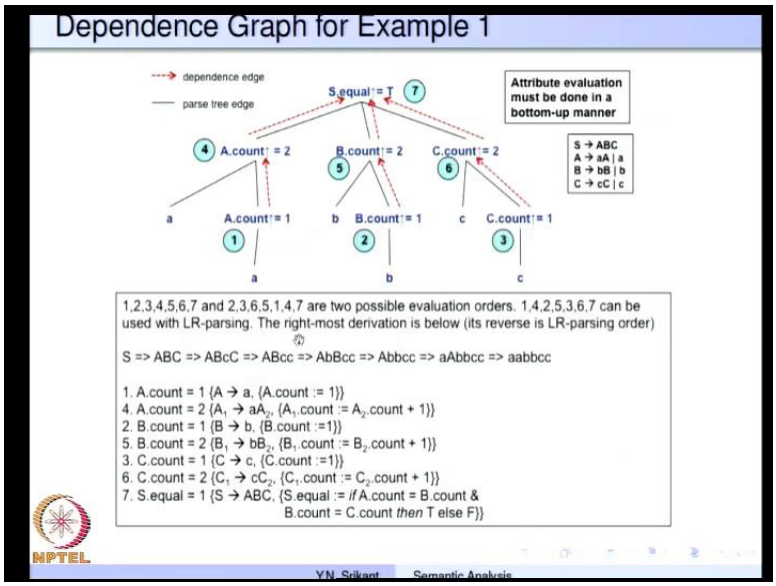
(Refer Slide Time: 45:56)



So, here is a formal algorithm which tells you the same thing. So, input is a parse tree T with unevaluated attribute instances and the output is T with consistent attribute values. So, let V comma E is equal to D G of T, so we have constructed the dependence graph. Now, initialize q W or a work list W with instances of attributes b such that b is in V and the in degree of b is o; that means you do not require any other attribute to compute b. So, these are the ones which can be computed first, so these are the lowest level in the parse tree as we saw before.

So, while W not equal to phi do, this is the very general algorithm, so it holds for any non circular attribute grammar. Remove some attribute b from the q W, the value of b is value defined by the appropriate attribute computation rule, this will become clear in the next example, now that we have computed the value of b. So, for all b comma c in E; that means, b is required for computation of c. So, you reduce the in degree of c by 1 and if in degree has become 0, then add c to the q. So, now c is eligible for you know to compute, so it gets its chance in the q sometime. The order in which we compute the attributes you know once they are in the q is not very important any order will do.

(Refer Slide Time: 47:45)



So, let us take the same example and understand the evaluation possible you know in a better way. So, there are many attribute evaluation orders possible here, so 1, 2, 3, 4, 5, 6, 7, so this is 1, 2, 3, 6, 5, 1, 4, 7, so 2, 3, 6, 5, 1, 4, 7. So, we never violate the dependences and then 1, 4, 2, 5, 3, 6, 7, is another 1, 4, 2, 5, 3, 6 and then 7.

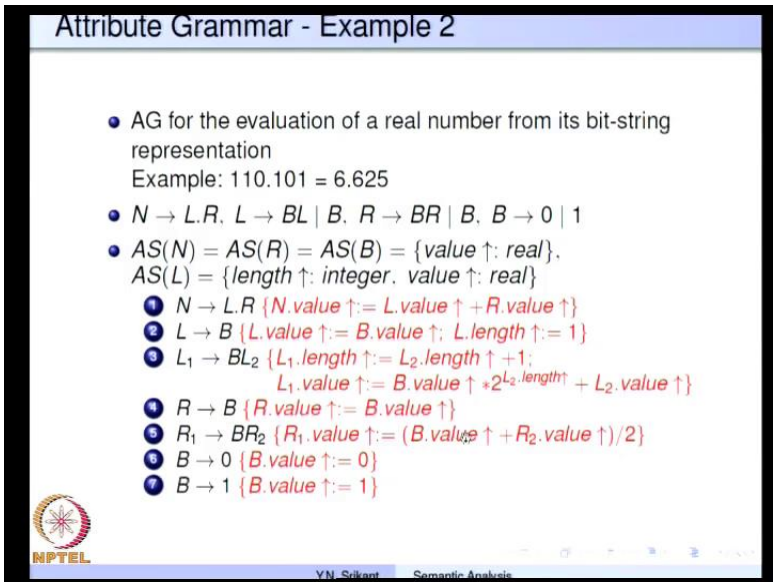
So, this is an order which can be used with l r parsing, really speaking we do not have to construct the parse tree explicitly in this case, because the parsing order is actually the order in which the attributes are also evaluated and we will never visit that part of the tree again. So, if we use the order 1, 4, 2, 5, 3, 6 and then 7 we can do it using the l r parsing strategy that is because the parser you know goes in this way. So, it shifts a on to the stack, then shifts this a on to the stack, reduces to this A, then these two together are reduced to this A and then it does the same with these two b and gets this capital B, so similarly these two c and then it gets this capital C. So, now, it reduces to S and winds up.

So, as we go along we are really traversing bottom up upwards and we will never go down again. So, it is possible to do the attribute computation as we go along and store the attribute value in the record of the non terminal on the stack itself. So, here is the right most derivation and the reverse of this is used by the l r parser. So, let us understand how the attribute computations happen in this case. So, node number 1 which

is here, so A dot count equal to 1, so that is the evaluation, the A to a is the production and A dot count equal to 1 is the rule.

So, similarly for this particular node the value of A dot count is 2 and that is obtained because of this production and its associated computation rule A 1 dot count equal to A 2 dot count plus 1. So, this is the order in which we go ahead. So, here this is quite clear and you can go through this example in detail later.

(Refer Slide Time: 50:35)



Here is another example very interesting one, attribute grammar for the evaluation of a real number from its bit string representation. So, for example if you have a binary string here 110.101. So, here is the binary point corresponding to a decimal point. So, 1 1 0 is 6 that is very clear and this 1 0 1 the weights actually go in this direction, so the first one has minus 1 weight, second one has minus 2, third one has minus three. So, 2 to the power minus 1 is 0.5 and 2 to the power minus 3 is 0.125. So, the value 0.625, so 6.625 is the value of this particular string.

So, if you consider the context free grammar which generates bit strings of this kind at the highest level you would have N going to L dot R, L generates the left part, R generates the fraction. So, L goes to BL or B, so just like the count of the generator a counts, here it generates bits. So, any number of B here and each B can go to either 0 or 1.

Similarly, R also generates any number of B here with each B going to either 0 or 1. So, now, we have a context free grammar, which generates strings on this side and strings on this side and joins them with a dot here, right? So, given this context free grammar here is an attribute grammar which evaluates the string and gets you this particular value, again this attribute grammar requires only synthesized attributes. So, A of the non terminal N is the same as the synthesized attributes of R and that is in turn same as the synthesized attributes of B. So, all these three have the same name attribute called value which is from the domain of real numbers. Whereas, the non terminal L has two attributes, one is length which is an integer and another called value which is a real number.

So, the basic idea is, as we generate these B, so this L would have accumulated certain value and B is placed to the left. So, therefore, the B should get a value which is equal to the length of the bit string here which is the length of L. So, that is why we require both length and value for this non terminal L. Similarly, R generates B R, so this B is generated fresh, R would have some value of its own, but since this is on the right hand side of the dot it implies that the value of R is now reduced by 2 and the new value is assigned as the sum of the values of B and R, so let us go through it and see. So, N going to L dot R the value of N is nothing but L dot value plus R dot value that is very simple.

If L generates a single bit B, L dot value is B dot value and L dot length is 1, so that is very simple again, because whatever b is either 0 or 1 is the value of L. If L 1 goes to B L 2, then L 1 dot length is L 2 dot length plus 1, which is very simple, because we are generating a new bit here and L 1 dot value is B dot value, this is a new bit position into 2 to the power L 2 dot length, this is what I was telling you just now L has some number of bits already generated and plus L 2 dot value, so this is the new value of L 1 here.

Similarly, R to B is R dot value equal to B dot value which is simple either 0 or 1 and R 1 going to B R 2, so this becomes R 1 dot value equal to B dot value plus R 2 dot value by 2. So, here this would have generated you know certain number of bits here and we have actually now said a new bit is being generated, R has a certain value which is already old.

So, take the value of R add the value of B and divide the whole thing by 2, because B now has a value which is always 0.5, the maximum value of the bit after the dot is 0.5, so

that is the reason why we say B dot value plus R dot value by 2. So, B to 0 and B to 1 have B equal to 0 and B dot value equal to 1. So, this is the attribute grammar as far as evaluation of real numbers from the bit sting is concerned. So, we will stop here and in the next class we will consider the parse trees for this particular attribute grammar.

Thank you.