

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

Lecture - 13
Semantic Analysis with Attribute Grammars Part – 2

(Refer Slide Time: 00:21)

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

NPTEL

Y.N. Srikant Semantic Analysis

Welcome to the lecture part 2 of Semantic Analysis. So, in this lecture we will continue with our discussion on attribute grammars, and attributed translation grammars.

(Refer Slide Time: 00:28)

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes*
- Two types of attributes: *inherited* and *synthesized*
- Each attribute takes values from a specified domain
- A production $p \in P$ has a set of attribute computation rules for
 - synthesized attributes of the LHS non-terminal of p
 - inherited attributes of the RHS non-terminals of p
- Rules are strictly local to the production p (no side effects)

NPTEL

Y.N. Srikant Semantic Analysis

So, to do a bit of recap, attribute grammars are extended context free grammars, every symbol of the set $N \cup T$ has associated with it a few attributes. There are two types of attributes inherited and synthesized of course, the same attribute cannot be both inherited and synthesized, but there could be you know several inherited and several synthesized attributes associated with each symbol. Each attribute takes a value from a specified domain, such as integer or real or cross product of these etcetera, etcetera.

And we associate attribute computation rules with each production, so specifically we associate rules for the computation of synthesized attributes of the left hand side non terminal, and we associate rules for the computation of inherited attributes of the right hand side non terminals of the production. Of course, the most important aspect of an attribute grammar is that it is strictly the rules strictly local to each production P there are no side effects in the rules.

(Refer Slide Time: 01:52)

The slide is titled "Synthesized and Inherited Attributes" and contains the following bulleted list:

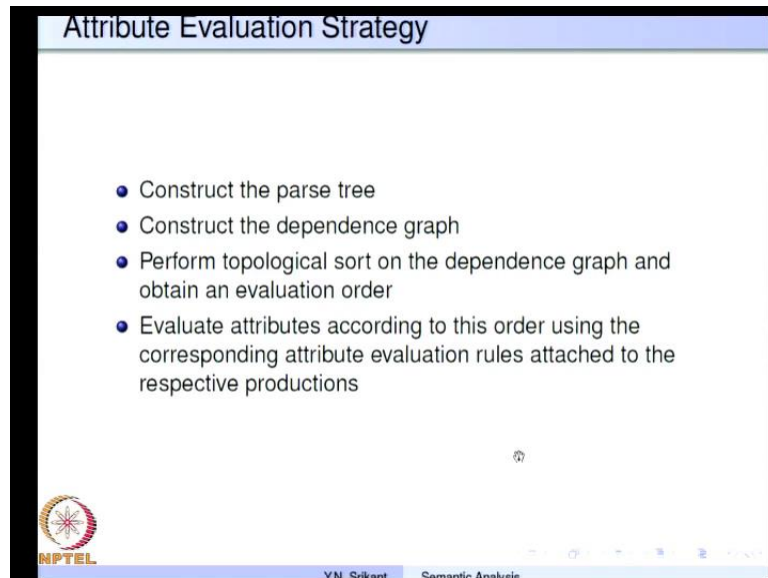
- An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- Synthesized attributes are computed in a bottom-up fashion from the leaves upwards
 - Always synthesized from the attribute values of the children of the node
 - Leaf nodes (terminals) have synthesized attributes (only) initialized by the lexical analyzer and cannot be modified
- Inherited attributes flow down from the parent or siblings to the node in question

The slide also features the NPTEL logo in the bottom left corner and a footer with the text "Y.N. Srikant Semantic Analysis".

The attributes of symbols are evaluated over a parse tree by making passes over the parse tree, it is possible to have more than one pass and on each pass there would be at least one attribute computed at the nodes. So, the synthesized attributes are computed in a bottom up fashion from the leaves upwards, so basically they are synthesized from the attribute values of the children of the node in the parse tree. Leaf nodes that is the terminals, they have only synthesized attributes and these are initialized by the lexical analyzer and of course, they cannot be modified. As far as the inherited attributes go

these actually the values flow down from the parent or the siblings to the node in question and then of course, they flow down words again from that node for the down.

(Refer Slide Time: 02:55)



The slide, titled "Attribute Evaluation Strategy", lists the following steps:

- Construct the parse tree
- 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


The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Semantic Analysis" in the bottom right corner.

A brief overview of the attribute evaluation strategy, we must build the parse tree, then we construct the dependence graph of the attributes. Perform the topological sort on the dependence graph and we obtain an evaluation order of course, then attribute evaluation is carried out using this order of course, then attribute evaluation is carried out using in this order and the attribute evaluation attached to the respective productions are used in the attribute evaluation process.

(Refer Slide Time: 03:31)

Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation
Example: 110.101 = 6.625
- $N \rightarrow L.R, L \rightarrow BL \mid B, R \rightarrow BR \mid B, B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value \uparrow: real\},$
 $AS(L) = \{length \uparrow: integer, value \uparrow: real\}$
 - 1 $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
 - 2 $L \rightarrow B \{L.value \uparrow := B.value \uparrow; L.length \uparrow := 1\}$
 - 3 $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$
 - 4 $R \rightarrow B \{R.value \uparrow := B.value \uparrow / 2\}$
 - 5 $R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$
 - 6 $B \rightarrow 0 \{B.value \uparrow := 0\}$
 - 7 $B \rightarrow 1 \{B.value \uparrow := 1\}$


Y.N. Srikant Semantic Analysis

So, we look at the example continue looking at the example that I showed you last time, so this is the attribute grammar for the evaluation of a real number from its bit string representation. For example, if you have a bit string 110.101, its decimal value is 6.625, here is a context free grammar N going to L dot R , L generates several bits on the left side of the dot. And in the grammar L going to B L or B , similarly R generates the bits on the right side of the dot and the grammar is R going to B R or B B is of course, a bit either 0 or 1.

Now, the attributes of the various, so if look at this string it generates this grammar generates binary strings with a dot in the middle. But, we have no indication of what its decimal value is, the attribute grammar is supposed to give us the decimal value of this any particular bit string generated by this grammar. The non terminals N R and B have just one synthesized attribute value, which is from the domain of real's, whereas the non terminal L has a synthesized attribute length and has another synthesized attribute value.

So, length is from the domain of integers and real is from the domain of real numbers, why do we require two attributes for L , whereas one's supplies us for N L R and B . So, to convince ourselves that this is required, let us just take a look at the strings for the left hand side of the dot, you know suppose we consider this one of the string 110. The you know power that is given to rather the exponentiation value given to this particular one, that is 2 to the power 2.

So, the first one is a $0 \cdot 2^0$, second one is 2^1 and the third one is 2^2 , this actually is based on the number of bits to its right side till the dot. So, there are 2 here, if we had more that would be the power of the base 2, this cannot be determined and if look at the grammar for L this b indicates this one that we are considering. And L to the right of it indicates the rest of the bits to the right of this particular one in this case.

So, without knowing the length of the string which is generated by L, it is not possible to assign the appropriate weight to this particular bit. In the case of the fraction the weight actually starts with minus 1 right after the dot, so this is 2^{-1} , this is 2^{-2} and the third bit is 2^{-3} . And the grammar is also such that, the B, which is nearest to the dot get generated first and R is the rest of the dots after the B.

So, therefore, assigning a weight to B is not at all difficult and it is sufficient to have just value as the attribute of this non terminal. So, let us look at the attribute grammar N going to L dot R, so the value of N is; obviously, the value of L plus the value of R, assuming that we assign weights to the bits in both L and R appropriately. So, L generates a bit by the production L going to B, so L dot value will be just B dot value either 0 or 1 and L dot length is 1 because, we are generating just 1 bit.

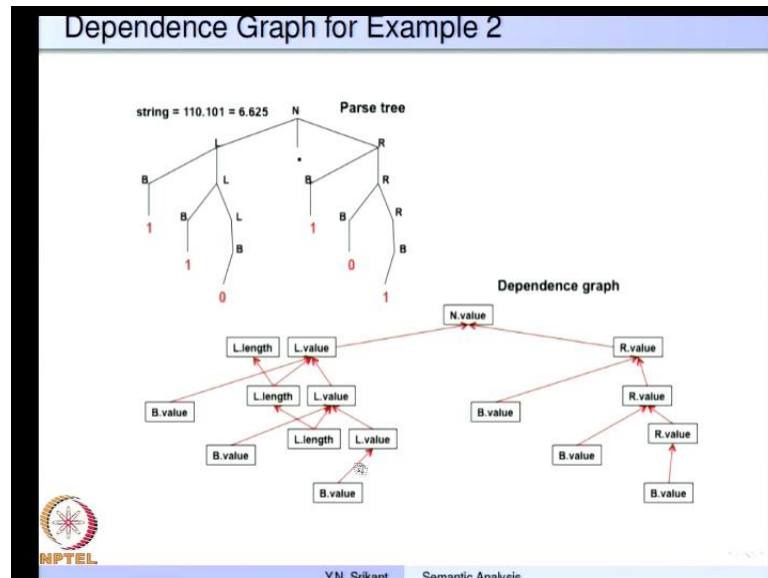
L₁ going to b L₂, so L₂ has certain length and B is an extra bit, which has now been generated. So, L₁ dot length is; obviously, L₂ dot length plus 1, what about the value L₁ dot value is; obviously, we take the L₂ dot value just like we take 2^2 plus the value of 10 which is 2 that is total will be 6. So, similarly we take the value of L₂, so L₂ dot value and we take the value of the bit B dot value either 0 or 1 multiplied by the appropriate weight $2^{\text{L}_2 \text{ dot length}}$.

So, here L₂ dot length would have been 2, so this would have been assign the value 2^2 to the power 2. So, that is how the value of L₁ gets, you know computed R to B is very simple R dot value equal to B dot value slash $2^{\text{R}_1 \text{ going to B}}$ and R₁ going to B R₂, so there is reason we have B dot value slash $2^{\text{R}_1 \text{ going to B}}$ is that we generate the, you know first bit after the dot gets away $2^{\text{R}_1 \text{ going to B}}$ so the bit value divided by $2^{\text{R}_1 \text{ going to B}}$ is R dot value.

Now, R going to B R₂ R₁ dot value will be take B dot value plus R₂ dot value and the whole thing is divided by $2^{\text{R}_1 \text{ going to B}}$ that is fairly easy to see. If you look at the production

because, R 2 had some value, now because of the bit it has been right shifted by 1 position. So, its value divided by 2 is the right and value for this particular R 2 and a new bit has been introduced, so its value is B dot value and as I said we must always divide the bit value by 2. Because, the weights begins with minus 1 here, B to 0 is b dot value equal to 0 and B to 1 will be B dot value equal to 1.

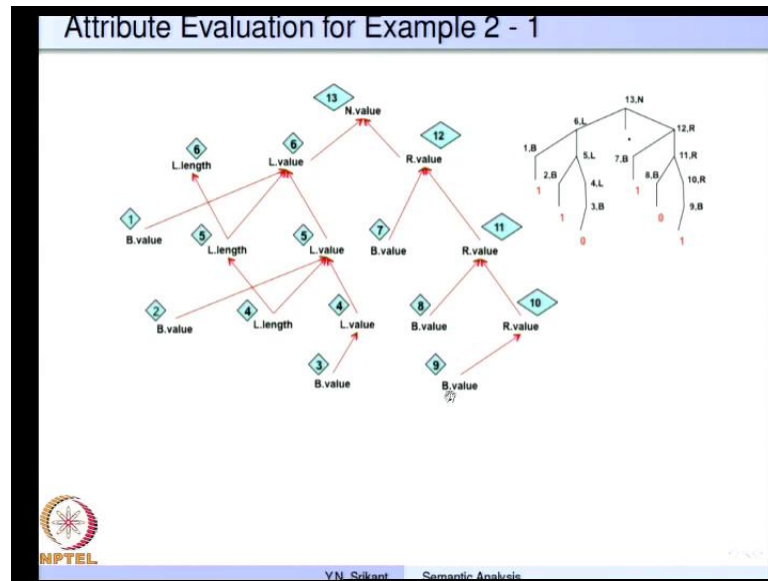
(Refer Slide Time: 10:09)



So, here is say a sample string 110.101 it is parse tree, the red ones are all the leaves and this is the dependence graph based on this parse tree and the attribute grammar. So, N dot value depends on L dot value and R dot value, then N dot length at this node depends on the L dot length below, whereas the L dot value here depends on B dot value L dot length and L dot value at the lower level. So, here in this case we are looking at this node, so L dot value here will depend on B dot value, then the L dot length and L dot value at this point.

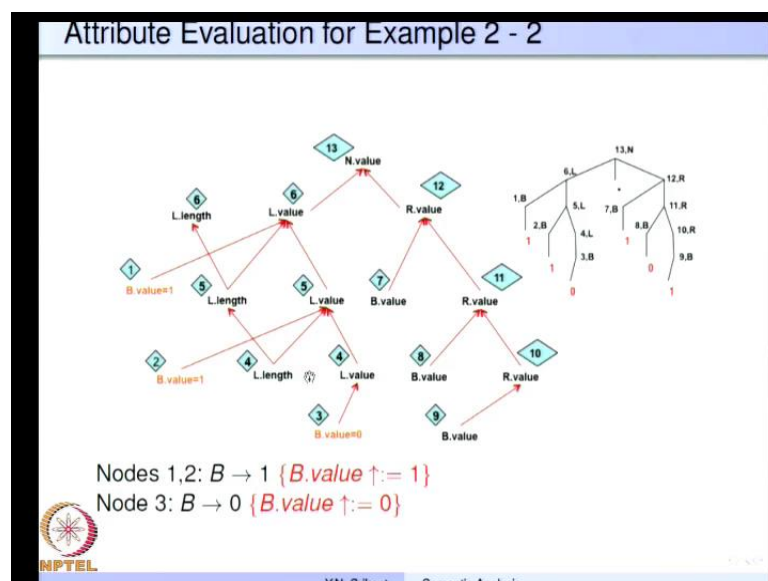
So, similarly for the other nodes of the tree as well and similar for the R part R dot value depends on B dot value and R dot value, so this recursively downwards. So, let us see how the attribute evaluation takes place given this particular dependence graph, a topological sort of the dependence graph makes these leaves to be evaluated first and then the next level and then the next level and so on and so forth.

(Refer Slide Time: 11:33)



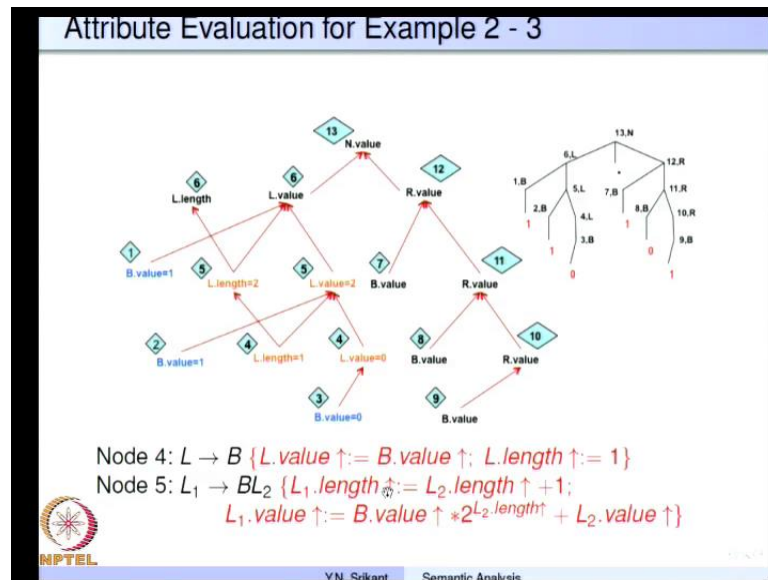
Here is the same graph, return you know in a slightly different way with the nodes number according to the sequence of evaluation. So, 1, 2 then 3 these are the 3 nodes which are evaluated first because, do not require any other node to be evaluated, so it is possible to evaluate L dot length also along with these 2 or rather these 3 because, it does not require anything else at this point of time, but we cannot evaluate L dot value because, L dot value requires b dot value. Therefore, we would rather wait until the B dot values are all completed and then go to this node 4, which has these 2 attributes and evaluate both of them at the same time.

(Refer Slide Time: 12:32)



So, the as I said node 1 is evaluated first, then node 2 and then node 3, just to save space I have showed all these 3 evaluations in the same slide. At nodes 1 and 2, we apply the production B going to 1 and the value obtain is B dot value equal to 1, at node 3 we apply the production B to 0 with a value B dot value equal to 0.

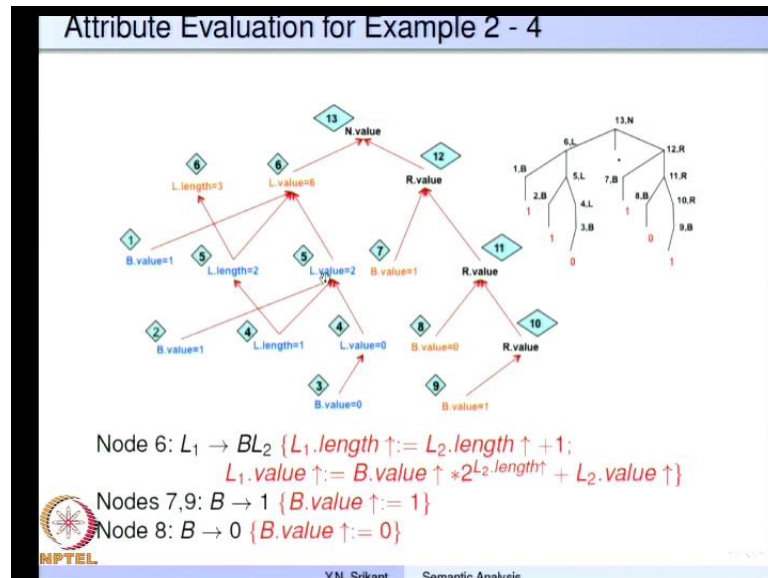
(Refer Slide Time: 13:04)



And then, so these 3 we were evaluated already, now it is the turn of these nodes to be evaluated. So, now L dot length equal to 1, L dot value equal to 0, so 4 and then 5 and you know these are the nodes, which will be evaluated, at node 4 the production even though all those values as I said node 4 is evaluated first and then node 5 because, the values of node 4 are required for node 5. So, at the node L of a 4 L to B is applied and the value can be computed using the production rule, so L dot value equal to B dot value and L dot length equal to 1.

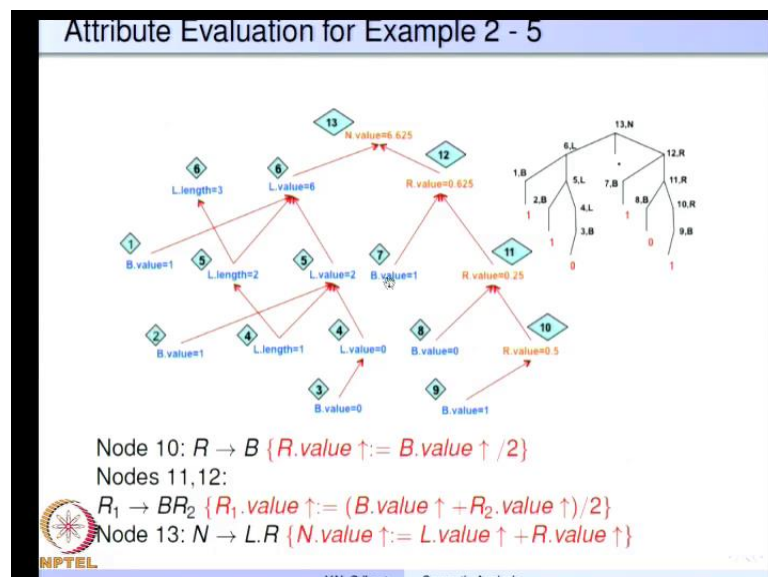
So, that appropriately uses 1 and 0 here, as we go upwards the production applied is L 1 going to B L 2. So, length gets incremented, so this becomes 2 and L 1 dot value is computed using the rule B dot value into 2 to the power L 2 dot length plus L 2 dot value. So, L 2 dot value is 0, L 2 dot length is 1, so this becomes 2 to the power 1 into 1 that is S 2.

(Refer Slide Time: 14:28)



So, now we go further, now node 6, node 7, 8 and 9 in that order will be evaluated, at 6 we have the same L 1 going to b L 2 and therefore, the value obtained here would be 2 plus 2 square. So, that would be 6 and length becomes 3 because, we you are generated 3 bits, so for at nodes 7 and 9, so 7 and 9 we apply B going to 1 and at node 8 we apply B going to 0, so the value here B is 1 and these 2 are this is 0 and these 2 are 1.

(Refer Slide Time: 15:09)




Then we evaluate the nodes 10, 11 and 12 and 13 in that order, so this gets R dot value equal to 0.5, because of the rule R going to B, this gets the value R dot value equal to

0.25. Because, this is a 0 this gets divided by 2 and then this gets you know the value 0.625 because, this is a 1 which gives you 0.5 and this is 525 finally, adding up the value from the left side and the right side, we get N dot value equal to 6.625. So, this is the order in which the evaluation happens over the parse tree and over the dependence graph.

(Refer Slide Time: 15:58)

Attribute Grammar - Example 3

- A simple AG for the evaluation of a real number from its bit-string representation
 Example: $110.1010 = 6 + 10/2^4 = 6 + 10/16 = 6 + 0.625 = 6.625$
- $N \rightarrow X.X, X \rightarrow BX \mid B, B \rightarrow 0 \mid 1$
- $AS(N) = AS(B) = \{value \uparrow: real\}$,
 $AS(X) = \{length \uparrow: integer, value \uparrow: real\}$
 - 1 $N \rightarrow X_1.X_2 \{N.value \uparrow := X_1.value \uparrow + X_2.value \uparrow / 2^{X_2.length}\}$
 - 2 $X \rightarrow B \{X.value \uparrow := B.value \uparrow; X.length \uparrow := 1\}$
 - 3 $X_1 \rightarrow BX_2 \{X_1.length \uparrow := X_2.length \uparrow + 1;$
 $X_1.value \uparrow := B.value \uparrow * 2^{X_2.length \uparrow} + X_2.value \uparrow\}$
 - 4 $B \rightarrow 0 \{B.value \uparrow := 0\}$
 - 5 $B \rightarrow 1 \{B.value \uparrow := 1\}$



Y.N. Srikant Semantic Analysis

So, let us take another example, so this particular example is again for computation of a real number from bits string representation, but the method is very different. So, look at the grammar, the previous grammar had L dot R and two different grammars for L and R productions for L and R. But, here we have just one non terminal X dot you know X and the production is N going to X dot X, X going to B X or B B going to 0 or 1, so when the non terminal X produces bits it is not possible to know, whether we are on the right side of the dot or on the left side of the dot.

So, because of this problem we cannot use the same attribute grammar as we had studied before, the strategy is going to be very different. But, in some sense it is simpler strategy as well, we compute the value of the string as it is, so we really do not worry about the dot at the beginning. So, 110 has value 6 and 1010 has the value 10 decimal value 10, now there are 4 bits after the dot, so length of the string here is 4, so we simply divide the fraction part by 2 to the power 4 and that gives us 6 plus 10 by 16, which is 6 plus 0.625 thereby we get the old value 6.625.

So, this is inherent you know the value inherent even if we had 0 extra here or 0 on the left side of this etcetera, etcetera. Let us take the case of this becoming 10100, so in that case the value is really double of 10 that is 20, but at the same time, the number of bits also becomes 5. So, instead of 10 by 2 to the power 4 we really have 20 by 2 to the power 5, which is same 10 by 2 to the power 4 and the value remains 6.625, here again we require, you know the value as an attribute of both N and B.

And as for as thus non terminal X is concerned, we require the length and also the value, so that is very easier to see, this is the value and this is the length. But, in the computation of the value we are not going to differentiate between the fraction part and the integer part. So, N going to X dot X we write it as X 1 dot X 2 just to differentiate between the 2two instances of the X, so N dot value will be X 1 dot value plus X 2 dot value divided by 2 to the power X 2 dot length.

So, x 2 dot value divided by 2 to the power X 2 dot length is what we have done here, X to B straight forward we get X dot value equal to B dot value and X dot length equal to 1. X going to B X will give us X 1 dot length equal to X 2 dot length plus 1, which is very easy because, we have added a bit here and X 1 dot value is B dot value into 2 to the power X 2 dot length. So, that is also something from the previous grammar plus X 2 dot value, these two are straight forward as before.


So, this is attribute grammar a different one for the computation of the real number from a bit string representation. The moral of this these two examples is that, the sentences in the language may be the same, the context free grammars may become different and there by the attribute grammars for the computation of the same value may also have to become different.

(Refer Slide Time: 20:19)

Attribute Grammar - Example 4

- An AG for associating *type* information with names in variable declarations
- $AI(L) = AI(ID) = \{type \downarrow: \{integer, real\}\}$
 $AS(T) = \{type \uparrow: \{integer, real\}\}$
 $AS(ID) = AS(identifier) = \{name \uparrow: string\}$
 - 1 $DList \rightarrow D \mid DList ; D$
 - 2 $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
 - 3 $T \rightarrow int \{T.type \uparrow := integer\}$
 - 4 $T \rightarrow float \{T.type \uparrow := real\}$
 - 5 $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
 - 6 $L_1 \rightarrow L_2 . ID \{L_2.type \downarrow := L_1.type \downarrow; ID.type \downarrow := L_1.type \downarrow\}$
 - 7 $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

Example: *int a,b,c; float x,y*
a,b, and c are tagged with type *integer*
x,y, and z are tagged with type *real*



Y.N. Srikant Semantic Analysis

So, let us move on let us take a different example, an attribute grammar for associating type information with the names in variable declarations. So, we have the grammar is given here, so let us see how it works DList is either D or DList semicolon D, so here D generates a single declaration and DList generates a list of declarations within D we have D going to T L. So, T is the type, so either integer or float and L is a list of names of this particular type, so example is given here integer a, b, c float x, y etcetera, etcetera, so integer a, b, c is generated by D going to T l.

Similarly, float x y is generated by D going to T L, but these two together will be generated by DList going to DList semicolon D and then the DList generating another D. So, that is how these two declarations get generated, now coming to the attributes of this grammar, here for the first time we introduce inherited attributes. So, for example, the non terminal L and the non terminal I D both of them have the type information has an inherited attribute.

Let us assume a user defined scalar type, you know integer comma real, which is permitted in C, C plus plus, Pascal etcetera. So, type is from the domain of two quantities integer comma real and these are inherited attributes of L and D, where as the non terminal T has a synthesized attribute type, which is again of type integer or real. The non terminal I D also has a synthesized attribute, which is the name of the variable and

the terminal symbol identifier also has name as it is synthesized attribute, which is nothing but a string of characters.

So, how does this particular attribute grammar work, you know as we go on for example, lets a take an example integer a, b, c. So, the type of the 3 variables a, b, c is integer a is of type integer, b is of type integer c is also of type integer, so these names a, b and c are really generated at this level by the production I D going to identifier. To associate the name with it is type we must actually take this particular integer and make it available to the list of names that is being generated.

So, that is precisely what we do here, the first production first two DList going to D or DList going to DList semicolon D have no computation associated with them, D going to T L has a attribute computation rule. So, it says L dot type that is the type of the names generated by L is nothing but T dot type, so L dot type is inherited and T dot type is synthesized. So, this things and synthesized and coming into L we will see an example with a parse tree within a short file T going to integer.

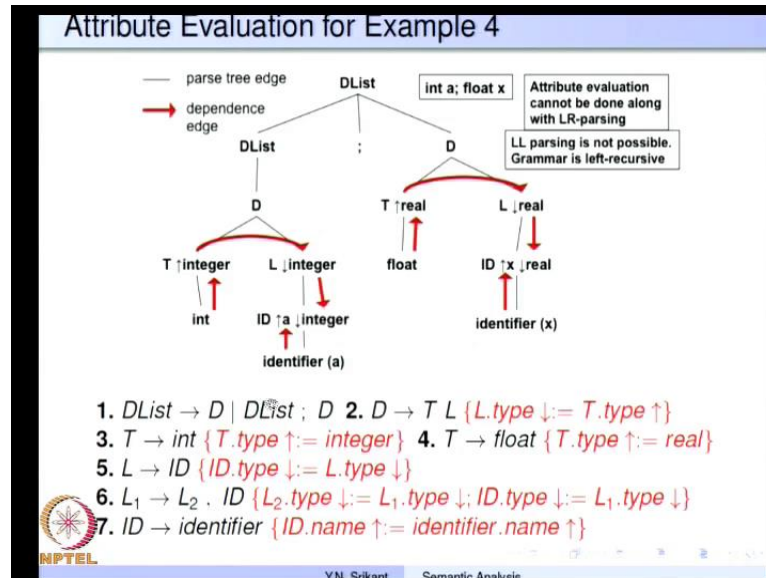
So, here T dot type is integer, so that's very simple, similarly T going to float gives us T dot type equal to real, L going to I D. So, again L has type as an inherited attribute and that has to be passed on to the I D has an inherited attribute, so I D dot type equal to L dot type, L 1 going to L 2 comma I D. So, the type information which is given to L 1 from the top is passed on to both L 2 and I D here, so therefore, L 2 dot type equal to L 1 dot type and I D dot type equal to L 1 dot type I D going to identifier, so I D dot name is identifier dot name.

So, at the level of I D we have associated both the type and the name of that particular variable. So, a b and c are tagged with type integer x, y, z are tagged with type real in this particular example, so we must observe here that the notation for inherited attributes is the attribute and then a down arrow. And we have attribute rules computation rules for the inherited attributes on the right hand side of the production, so L has inherited attributes, so we provide a rule computation for it.

But, we do not provide any rule of computation for the synthesized attribute of T because, it is on the right hand side of this particular production. Whereas, T is on the left hand side of this production, so we provide a rule of computation for it, so similarly L 1 going to L 2 comma I D L 2 dot type is inherited. So, there is a rule of computation I

D dot type is inherited, so there is a rule of computation. But, L 1 dot type is also inherited, so and it is on the left hand side of the production, so we do not provide any rule of computation for L 1 dot type.

(Refer Slide Time: 26:38)



So, let us take this example and understand how the inherited attributes flow over the parse tree. So, the red edges are all attribute dependence edges and the black edges are all parse tree edges, so the sentence is integer a semicolon float x, so DList gives us DList semicolon D, this D generates float x and this D DList generates D and then that D generates integer a. So, to begin with of course, at this level DList going DList semicolon D there is no contribution, rather no attribute computation here.

The first attribute computation is associated only with the, so let us take this first it is simpler. So, we have T and then that goes to float and the here L it goes to ID and ID goes to identifier, the identifier is x, so here float is a synthesized attribute and it is flows to the node T and the rule of computation is T going to float. So, T dot type gets real, so that is how this attribute real is computed, here the string x flows to ID, so ID going to identifier and computation is ID dot name equal to identifier dot name, so that is how this x gets computed at this particular node.

Now, the attribute real which is synthesized by T is passed on as an infinite attribute to L and that happens in the role D going to T L. So, we have D going to T L here, so L dot type is T dot type, so that is the dependence and the computation associated with this


production. So, similarly L going to I D will pass on the attribute from L to I D, so L going to I D, so I D dot type equal to L dot type, so the dependence is correct, the same thing holds here as well.

So, integer becomes computes into integer at T and then T flows in as an inherited attribute to L and that flows into I D as an inherited attribute, the name a flows as a synthesized attribute to the non terminal D. So, the flow of attributes here, so there are two nodes, which are provided here, number one attribute evaluation cannot be done along with L R parsing that is because, there are inherited attributes here and that will become clear as we go along. L L parsing of this grammar is not possible because, the grammar left recursive, so this is also necessary, because we are going to see how to modify this grammar later to enable L L parsing.

(Refer Slide Time: 29:52)

Attribute Grammar - Example 5

- Let us first consider the CFG for a simple language
 - 1 $S \rightarrow E$
 - 2 $E \rightarrow E + T \mid T \mid \text{let } id = E \text{ in } (E)$
 - 3 $T \rightarrow T * F \mid F$
 - 4 $F \rightarrow (E) \mid \text{number} \mid id$
- This language permits expressions to be nested inside expressions and have scopes for the names
 - $\text{let } A = 5 \text{ in } ((\text{let } A = 6 \text{ in } (A * 7)) - A)$ evaluates correctly to 37, with the scopes of the two instances of A being different
- It requires a scoped symbol table for implementation
- An abstract attribute grammar for the above language uses both inherited and synthesized attributes
- Both inherited and synthesized attributes can be evaluated in one pass (from left to right) over the parse tree
- Inherited attributes cannot be evaluated during LR parsing


Y.N. Srikant Semantic Analysis

So, another example of attribute grammars and this is a much more complicated example, here is a fairly simple language of expressions S going to E, E going to E plus T plus or T or a new type of expression let i d equal to E in E. Then, we have as usual T going to T star F or F, F going to parenthesis E parenthesis or number or i d. So, the novel T and the specialty of this particular language is, we are permitted to define a value E for this particular name i d and that binding can be used within this expression E.

So, this can be nested as you can see E can again be a let i d expression and this also can be a let i d expression recursively. So, this language permits expressions to be nested

inside expressions and have scopes for the names, so let me show you with an example what we mean by the scope of this particular name i , in this particular production E going to let i equal to E in E . Here is the example, let A equal to 5 in now a , so we have, so far said let i equal to E , so i is A E is 5.

And then we have a new expression this entire bracket a term, so that is a second E , the second E has similar structure let A equal to 6 in again a new expression begins A star 7. And that completes and then we have a minus A , so this entire thing let A equal to 6 in A star 7 is the part of E here and then we have you know A which is the second part for this the scopes of variables A in both cases are shown in different colours. So, this A equal to 5 is valid for this particular A only, where as within this parenthesized expression this A equal to 6 is valid here at this point of the expression.

So, the meaning is this A takes the value of 6, so giving us 42 and then this entire expression let A equal to 6 in A star 7 has the value 42. So, now, this particular A equal to 5 is valid for this A , so this takes the value 5, this entire expression took the value 42. So, 42 minus 5 is 37, so the scope of this A and this A are clearly different, so the same name A , but 2 different values and 2 different scopes. Parsing and evaluating such expressions is again not trivial, it requires a the concept of a symbol table to make it happen, we are going to see that as well.

So, what we are going to show is an abstract attribute grammar for the above language, which uses both inherited and synthesized attributes. And these attributes can of course, be evaluated in one pass over the parse tree from left to right, I will show you an example of how that can be done as well. Inherited attributes cannot be evaluated during LR parsing, so this is a general rule unless we place a large no of restrictions on the inherited attributes this rule holds. Therefore, whenever we use inherited attributes the assumption is that we use LR parsing. And whenever there are only synthesis attributes, then the grammar can be used along with LR parses.

(Refer Slide Time: 34:27)

Attribute Grammar - Example 5

- 1 $S \rightarrow E \{ E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow \}$
- 2 $E_1 \rightarrow E_2 + T \{ E_2.symtab \downarrow := E_1.symtab \downarrow; E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow \}$
- 3 $E \rightarrow T \{ T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow \}$
- 4 $E_1 \rightarrow \text{let } id = E_2 \text{ in } (E_3) \{ E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow; E_3.symtab \downarrow := E_1.symtab \downarrow \setminus \{ id.name \uparrow \rightarrow E_2.val \uparrow \} \}$
- 5 $T_1 \rightarrow T_2 * F \{ T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow; T_2.symtab \downarrow := T_1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow \}$
- 6 $T \rightarrow F \{ T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow \}$
- 7 $F \rightarrow (E) \{ F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow \}$
- 8 $F \rightarrow \text{number} \{ F.val \uparrow := \text{number}.val \uparrow \}$
- 9 $F \rightarrow id \{ F.val \uparrow := F.symtab \downarrow \{ id.name \uparrow \} \}$

MPTEL
Y.N. Srikant Semantic Analysis

So, here is the attribute grammar for that particular context free grammar, the symbol table is an entity, which stores the names and their values. The symbol table, which is valid at a particular point in the expression is actually handed down to the expression from its parent in the parse tree. So, for example, in this case the symbol table which is valid within the expression E is phi because, this is the start symbol of the grammar.

So, E dot symtab equal to phi and that is E dot symtab is an inherited attribute and after the evaluation is completed E produces a value and E dot val is now assigned to S dot val and that is the value of the entire expression. Let us take this production E 1 going to E 2 plus T, so E going to E plus T and the instances numbered as 1 and 2, there was a symbol table available to E 1 and that is a same symbol table, which will be made available to both E 2 and T without any modification.

So, E 2 dot symtab is E 1 dot symtab, E 1 dot and T dot symtab is also E 1 dot symtab, so remember the order in which these attribute computation rules are written is not very important. Because, this is not the final order E 1 dot val is nothing but the sum of the values produced by this E 2 and this T, so E 2 dot val plus T dot val, so this is the value of E 1 dot val. So, E to T is trivial there is just a copy of the two attributes from E to T and so T dot symtab is E dot symtab and E dot val is T dot val.

The important things happen in production number 4 here and then it also happens in production number 9. So, let us take production number 4, let E going to let id equal to

E 2 in E 3, so the symbol table which was available to E 1 is made available to the expression E 2. So; that means, E 2 dot symtab is equal to E 1 dot symtab, it is also made available to E 3, so the symbol table of E 1 is made available to E 3 also, but it, so happens that this particular association of i d and E 2 dot value, now actually is introduced into the symbol table produced by E 1.

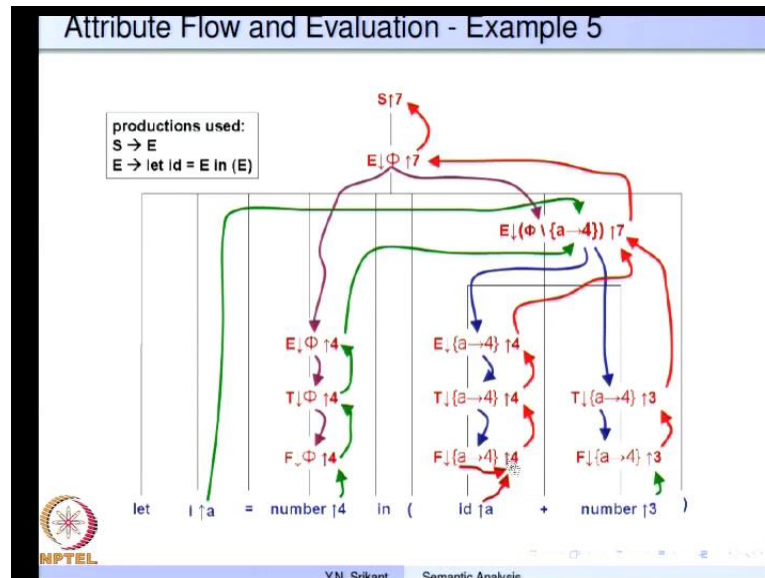
So, that is written as E 3 dot symtab is equal to E 1 dot symtab overridden by the entity or association i d dot name going to E 2 dot val. So, the consequence of this is twofold for example, if the name i d dot name, already you know is available in E 1, so there is another association of that particular name to some other value. Then that particular association is forgotten and a new association of i d to E 2 dot val is produced, so the old association is suppressed and the new association rules within the expression E 3.

I said carefully suppressed not deleted because, as soon as the scope of this particular end expression let i d equal to E 2 in E 3 is completed. The old value of the association from the name i d to some other value that was present in E 1 should be made available again. ((Refer Time: 38:55)) So, in this case for example, A equal to 5 is suppressed in A star 7 by the association of A to 6, but as soon as this expression is completed the value of A to 5 is available in this particular expression or the variable A.

So, this operator one as the overriding operator and not the deletion operator, T 1 going to T 2 star F is very similar to E 1 going to E 2 plus T and does not require too much elaboration. The symbol table is just copied T 2 dot symtab and F dot symtab or T 1 dot symtab and T 1 dot val is T 2 dot val star F dot val T 2 f is similar to E to t. So, and there is just copy of the attributes, F going to parenthesis E parenthesis also has just a copy of the attributes here, F 2 number says F dot val equal to number dot val and now F 2 i d again here is the usage of the variable ((Refer Time: 40:09)).

So, for example, in this A star 7 A is a usage of this particular definition and this A is a usage of this particular definition. So, that definition you know that usage would have been produced by F 2 i d, so F dot val is obtained by looking up the name i d dot name in F dot symtab, which is a F dot symtab is available here, as an inherited attribute and i d dot name is available as a synthesized attribute of the terminal. So, we look up this name in the symbol table and produce the value, which is assign to F dot val, so this looking up operation is indicated by the two brackets and name inside.

(Refer Slide Time: 40:57)



So, here is an example showing you how the flow of attributes takes place, the expression is let a equal to 4 in a plus 3, so a equal to 4, so a plus 3 becomes 7, so that is the value which is produced here that is correct. So, now, S going to E, so it initializes the symbol table to phi, so that is why the symbol table is shown as phi here, the notation used is slightly different here, the inherited attribute has it is arrow to the left side of the value.

And the similarly, the synthesis attribute has up arrow the left side of the value, both notations are used when the rules are written writing it the way I showed you before is usual practice and when we write it along with the non terminal this is the usual practice. So, this phi now is available in the non terminal E and at this point we have applied the production E going to let i d equal to number or rather expression in parenthesis etcetera, etcetera.

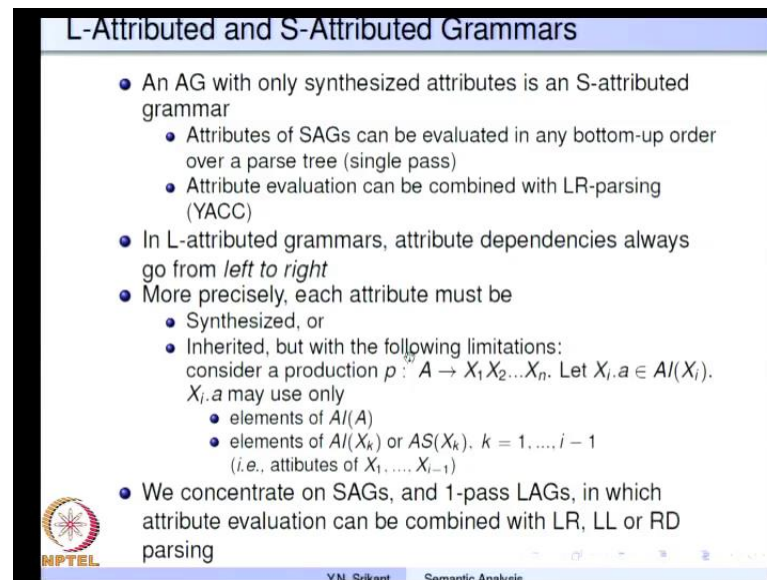
So, the production uses let i d equal to E in E at this point and as the rules indicate the symbol table phi is passed on to this E, which is the second one and the third E as well. So, this is the third E for both of them, the phi is made available, this phi is copied and sent down wards to T and F as indicated here and this name a is synthesized from here and is made available to this particular E. So, i d the a that I was that I have talked about here is nothing but the i d here, so this i d to E association is made available to this E.

So, let us see how that happens, the number 4, so a equal to 4 is the expression, so this E is nothing but just 4. So, that goes upwards to F and then T and then E as synthesized attribute and that E is made available to this particular E. So, a to 4 is the mapping or the association which is produced and that overrides whatever association was present in phi for the same name a, well in this case phi indicates nothing. So, a to 4 is only association now in the inherited attribute or the symbol table of this particular instances of E.

Remember, we have not yet produced this value 7, so this a to 4 is a new symbol table and that is made available to this E and this T because, the production that we have applied is E going to E plus T at this point in the syntax tree. So, this flows down words from E to T and T to f and here is the usage of a, so in this production in this expression E, so we must look up the name a in the symbol table and that is what we have done. So, the symbol table is a to 4, when we look up the name a we get the value 4 here and that value 4 passed on to the next level as the value of F.

So, this goes upwards as the value of T and this value is passed on as the value of E, similarly F 2 number produces 3 which is passed up words. And the rule the associated with E to E plus T combines 4 and 3 in a summation to produce the value 7, which is passed upwards as the synthesized attribute of E and that in turn goes to S, so this E as I told you is the this particular E. So, that is the value, which has produced 7 really these produce a 7 and that is the value which is sent upwards as the value of the entire expression. So, this is how the attribute evaluation happens the sequence in which it happens we will see in a while.

(Refer Slide Time: 45:43)



L-Attributed and S-Attributed Grammars

- An AG with only synthesized attributes is an S-attributed grammar
 - Attributes of SAGs can be evaluated in any bottom-up order over a parse tree (single pass)
 - Attribute evaluation can be combined with LR-parsing (YACC)
- In L-attributed grammars, attribute dependencies always go from *left to right*
- More precisely, each attribute must be
 - Synthesized, or
 - Inherited, but with the following limitations:
consider a production $p: A \rightarrow X_1 X_2 \dots X_n$. Let $X_i.a \in AI(X_i)$.
 $X_i.a$ may use only
 - elements of $AI(A)$
 - elements of $AI(X_k)$ or $AS(X_k)$, $k = 1, \dots, i - 1$
(i.e., attributes of X_1, \dots, X_{i-1})
- We concentrate on SAGs, and 1-pass LAGs, in which attribute evaluation can be combined with LR, LL or RD parsing

MPTEL
Y.N. Srikant Semantic Analysis

So, before we look at the sequence of attribute evaluations for the examples that I showed you. Let us look at a classification of attribute grammars called L attributed grammars and S attributed grammars, so if you have only synthesized attributes in an attribute grammar, then it is called as an S attributed grammar. So, it is a very simple definition, there are no inherited attributes at all and in the case of such an SAG any bottom up evaluation order over a parse tree can be used to evaluate the attributes.

And; obviously, a single parse over the tree is sufficient and it, so happens that attribute evaluation can be combined with L R parsing as well. Because, L R parsing produces one bottom up you know does a bottom up parse over the parse tree, rather the construction of the parse tree itself is done in a bottom up fashion it is quite, you know convenient to do the attribute evaluation also in a combined fashion along with L R parsing.

So, YACC permits only synthesized attributes and the rules that we write for YACC, you know YACC productions will be executed as we go up in the L R parsing process. I will show you some examples of this very soon, what about L attributed grammars, L attributed grammars they have their attribute dependencies go from left to write. So, the very precisely, so each attribute must be synthesized in which case there is no problem at all or it could be inherited and if it is inherited then there is a limitation, suppose the production is, so now, we formulize what is known as left to write dependence.

Suppose the production is $A \rightarrow X_1 X_2 \text{ etcetera } X_N$ and let the attribute $X_i \cdot a$ be an inherited attribute of the symbol X_i . So, if $X_i \cdot a$ is the attribute it may use only the inherited attributes of the non terminal A , the left hand side or the elements of A_i of X_k , which means the inherited attributes of the symbols to the left of i or the synthesized attributes of S_k , k going from 1 to $i - 1$. So, when we are looking at position $i - 1$ to $i - 1$ simply means the symbols to the left side of that particular non terminal X_i .

So, we can use the inherited or the synthesized attributes of the symbols to the left of X_i , so X_1 to X_{i-1} only. So, that is what we mean by the statement the dependencies go from left to write, so we will not be able to use any attribute of a symbol which is to the right of X_i . So, X_{i+1} cannot supply any attributes to X_i , if it does then it does not become L attributed, the advantage of L attributed grammars is that, we can make a series of left to write evaluations over the parse tree and evaluate all the attributes.


And if we further restrict the attribute evaluation, such that it can be done in 1 pass over the parse tree from left to write, then it is called as LAG 1 or 1 pass LAG. So, 1 pass LAG is the attribute evaluation can be very easily done with LL or you know recursive descent parsers for SAG is it can be done with LR, LL or RD parsing any of them. But, of course, if there are inherited attributes in general we cannot use LR parsing to parse to go evaluate the attributes.

(Refer Slide Time: 50:23)

Attribute Evaluation Algorithm for LAGs

Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values

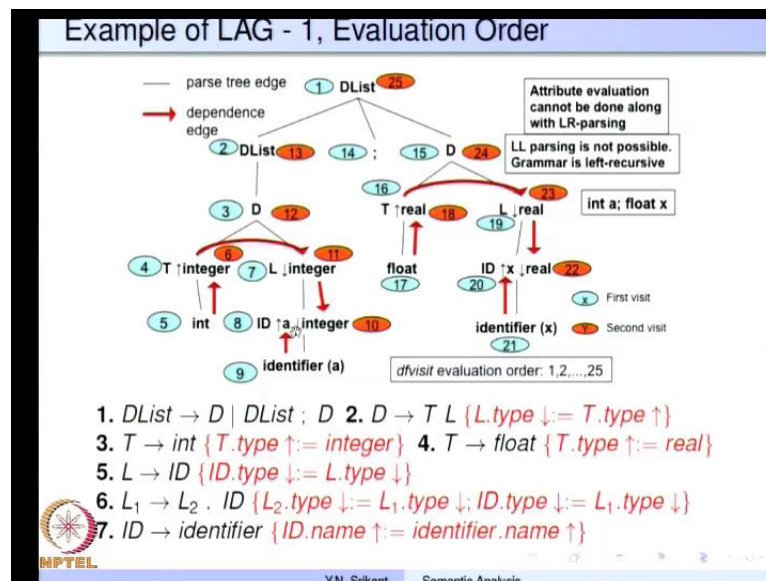
```
void dfvisit( $n$ : node)
{
  for each child  $m$  of  $n$ , from left to right do
  {
    evaluate inherited attributes of  $m$ ;
    dfvisit( $m$ )
  };
  evaluate synthesized attributes of  $n$ 
}
```

 Y.N. Srikant Semantic Analysis

So, how does one do attribute evaluation in the case of L attributed grammars or LAG's, so let us assume that a parse tree T with unevaluated attribute instances is given to us. And we are suppose to produce a parse tree T with consistent attribute evaluation, attribute values after the evaluation passes, so basically we do a depth first search or depth first visit on the parse tree. So, d f visit with the root n is the beginning point, for each child m of n from left to right do evaluate the inherited attributes of m, then do a d f visit on m.

So, first the on entry to a node we evaluate the inherited attributes, then do d f visit which intern goes down wards in the tree. And after we complete the evaluation of all the you know dependence of this m, we evaluate the synthesized attributes of n and then return to the higher level. So, first inherited then visit the node and then compute the synthesized attributed the node and then return, so this is the general procedure.

(Refer Slide Time: 51:51)



So, let us apply that procedure to our examples here, so this was the declaration example, so we have a you know this grammar is also L attributed. So, let us make sure of that first, so here L dot type equal to T dot type, so L dot type uses only the synthesized attribute of T. Here of course, we have only synthesized attribute, so there is no violation of any rule, the same is true for T to float as well L to I D we have I D dot type equal to L type, so I D dot type is an inherited attribute which uses the inherited attribute of L. So, no violation L 1 going to L 2 dot I D.

So, we L 2 dot type uses the parents attribute and I D dot type also uses parents attribute, so no violation and I D dot name is a synthesized attribute, identifier dot name is also synthesized, so there is no violation of any rule. So, this is a perfectly valid L attributed grammar, here is the evaluation order, so if you do a depth first search on this tree we begin at the root. So, the green once are all visits for the first time and the orange once are all visits of the same node a second time.

So, that a total of 25 you know visits are required, so and the d f visit evaluation order is 1, 2, etcetera up to 25. So; that means, we visit 1, then 2, then 3 and then 4, 5 then you know 6 here than 7, 8, 9, 10, 11, 12, 13, then we come you know to the semicolon even though we do not have anything to do we just have to visit it, then 15, 16, then 15, this is 18, this is 19, 20, 21, 22, 23, 24, 25. So, when we do this, let us see how the attribute evaluation takes place, so this is the first rule D going to T L at which an inherited attribute will have to be evaluated.

But, you know we really cannot L dot type equal to T dot type cannot be evaluated immediately. So, what we really do is we go down to 4 and then 5 and when we come back to 4 in the that is the visit number 6, we can compute the synthesized attribute of T. Then we are ready to compute the inherited attribute of this particular L, so that is possible, then we go down again passing down to I D, this identifier is then passed on to this. So, that is the end of this particular evaluation in this sub tree, the evaluation happens in this sub tree as well in a similar way.

So, basically we must do a d f visit and a teach node if the dependencies are satisfied we will be able to apply a production rather a computation rule, evaluate that particular attribute and then go further. So, here as we go on, so this the inherited attribute of the D is nothing whereas, for T there is no inherited attribute, so we go down and there is a synthesized attribute which can be evaluated, so on the return path, then just before visit to L we can you know evaluate the inherited attribute of L.

And then we go down to I D nothing to do and the inherited attribute of I D can be evaluated and when we return from here, the synthesized attribute of I D can be evaluated as well. So, this is how the attribute evaluations happen, thank you very much we will continue the lecture in the next part.

Thank you.