


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

**Lecture - 14**  
**Semantic Analysis with Attribute Grammars Part-3**

(Refer Slide Time: 00:20)

**Outline of the Lecture**

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


 Y.N. Srikant Semantic Analysis

Welcome to part three of the lecture on attribute grammars. We will continue with attribute grammars, attributed translation grammars, and semantic analysis in this lecture as well.

(Refer Slide Time: 00:29)

**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)

 Y.N. Srikant Semantic Analysis

So, again a brief recap on attribute grammars. So, attribute grammars are extensions of context free grammars. And with each symbol, either terminal or non-terminal symbol of the grammar, there is a set of associates or attributes which are associated with it. Two types of attributes are possible; inherited and synthesized. And there are rules associated with each of the productions to compute these attributes. So, synthesized attributes of the left hand side non-terminal, and inherited attributes of the right hand side non-terminals are provided with rules to compute the attributes. Rules are of course strictly local to the production and they have no side effects.

(Refer Slide Time: 01:22)

**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

NPTEL  
VN. Srikant    Semantic Analysis

So, now the classification of L-attributed and S-attributed grammars. SAGs are very simple. They are only synthesized attributes. And we can use any bottom-up evaluation strategy in a single pass to evaluate all the attributes. So, because of this property they can be combined with L R- parsing. And therefore, YACC is very useful in with such SAGs. In L-attributed grammars, the dependencies go from left to right. So, very specifically the attributes may be synthesized. And if they are inherited, they have the following limitation. So in a production  $p$ ,  $A$  going to  $X_1, X_2, X_n$ . Suppose, we consider an attribute  $X_i.a$  which is inherited, and then  $X_i.a$  may depend on only the elements of  $AI(A)$ ; that is the inherited attributes of the left hand side non-terminal. Or the attributes of any of the symbols to the left of  $X_i$ , that is  $k$  equal to 1 to  $i - 1$ . So, we will concentrate on the SAG and 1-pass LAG, in which we can do attribute evaluation with L R, LL or RD parsing.


(Refer Slide Time: 02:41)

### 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  $n$ 
}
  
```

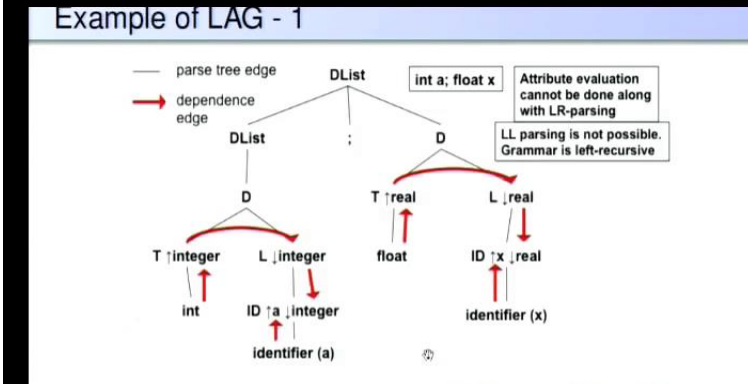


Y.N. Srikant    Semantic Analysis

So, attribute evaluation of LAGs is very simple. We just do that first search on the, you know, parse tree. So, given a parse tree with unevaluated attributes, the output is a parse tree with consistent attribute values. So, the method is very simple for each child  $m$  of  $n$ , from left to right. Do evaluate the inherited attributes of  $m$  because we are visiting  $m$  next; so  $dfVisit(m)$ . That is the recursive call. And after we visit all the children, evaluate the synthesized attributes of  $n$ . So, this is the algorithm for attribute evaluation of L attributed grammars.

(Refer Slide Time: 03:27)


### Example of LAG - 1



— parse tree edge  
 → dependence edge

**int a; float x**    Attribute evaluation cannot be done along with LR-parsing  
 LL parsing is not possible. Grammar is left-recursive

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.  $ID \rightarrow identifier \{ ID.name \uparrow := identifier.name \uparrow \}$



Y.N. Srikant    Semantic Analysis


So, this is the example we saw last time. It is indeed an LAG. So, there are no dependencies which go from right to left. So, for example in  $D \rightarrow T L$ ,  $L \text{ dot type}$  is equal to  $T \text{ dot type}$ . So,  $T \text{ dot type}$  is synthesized to the left of  $L$  and then that is inherited into the non-terminal  $L$ .

(Refer Slide Time: 03:51)

### Example of Non-LAG

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

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


Y.N. Srikant    Semantic Analysis

So, let us see what makes a grammar; a non- L attributed grammar or non-LAG as they say. So, it is actually a similar grammar; the associating type information with names in variable declarations. But, let us look at the syntax. So, examples are  $a, b, c : int$   $X, Y : float$ . In the previous grammar, we had  $int a, b, c$  and  $float x, y$ . Here, the type information comes after the variable list. So, this is a Pascal style declaration. The effect is similar.  $a, b$  and  $c$  are tagged with type *integer*. And  $x, y$ , of course there is no  $z$  here. This is a minor error.  $x$  and  $y$  are tagged with type *real*. So, in this case if you look at the production  $D$  going to  $L : T$ , all others are the same.

Now, the type information  $T$  is  $T \text{ dot type}$  and that is synthesized. And there is a non-terminal  $L$ , which has an inherited attribute  $L \text{ dot type}$ . So, we would like  $L \text{ dot type}$  to become  $T \text{ dot type}$ ; that means, the synthesized attribute actually flows to the left and gets into  $L \text{ dot type}$ . So, this is a right to left dependence and not a left to right dependence. And, this is exactly what makes the, you know, attribute grammar non-LAG. We cannot evaluate it in a left to right pass because when we reach  $L$ , the type information is not yet ready. And after we synthesize the type information, we have

already passed L. So, since we are looking at 1- pass evaluation, this attribute grammar is not LAG.


(Refer Slide Time: 05:52)

### Example of LAG - 2

- 1  $S \rightarrow E \{ E.syntab \downarrow := \phi; S.val \uparrow := E.val \uparrow \}$
- 2  $E_1 \rightarrow E_2 + T \{ E_2.syntab \downarrow := E_1.syntab \downarrow; E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.syntab \downarrow := E_1.syntab \downarrow \}$
- 3  $E \rightarrow T \{ T.syntab \downarrow := E.syntab \downarrow; E.val \uparrow := T.val \uparrow \}$
- 4  $E_1 \rightarrow \text{let } id = E_2 \text{ in } (E_3) \{ E_1.val \uparrow := E_3.val \uparrow; E_2.syntab \downarrow := E_1.syntab \downarrow; E_3.syntab \downarrow := E_1.syntab \downarrow \setminus \{ id.name \uparrow \rightarrow E_2.val \uparrow \} \}$

**Note: changing the above production to:**  
 $E_1 \rightarrow \text{return } (E_3) \text{ with } id = E_2 \text{ (with the same computation rules) changes this AG into non-LAG}$

- 5  $T_1 \rightarrow T_2 * F \{ T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow; T_2.syntab \downarrow := T_1.syntab \downarrow; F.syntab \downarrow := T_1.syntab \downarrow \}$
- 6  $T \rightarrow F \{ T.val \uparrow := F.val \uparrow; F.syntab \downarrow := T.syntab \downarrow \}$
- 7  $F \rightarrow (E) \{ F.val \uparrow := E.val \uparrow; E.syntab \downarrow := F.syntab \downarrow \}$
- 8  $F \rightarrow \text{number} \{ F.val \uparrow := \text{number.val} \uparrow \}$
- 9  $F \rightarrow \text{id} \{ F.val \uparrow := F.syntab \downarrow [id.name \uparrow] \}$



Y.N. Srikant    Semantic Analysis

another example of L attributed grammars. So, this is the expression evaluation with, you know, a nice construct such as L. Let id equal to expression in expression. So, this attribute grammar is indeed L attributed. So, if you look at any attribute rules, attribute computation rules, for example, here the inherited attribute E dot syntab of this is initialized. So, there is no violation of any rule here and S dot val is synthesized and that becomes E dot val.

So, there is no violation of any rule here, regarding LAG s. In this case, the inherited attribute from E one comes into E two and T. So, that is perfectly valid. So, there is no violation here either. Same is true about E dot T as well. And, also T one to T two star F, T to F, F to E. All these are similar. So, let us look at the important production E going to let id equal to E in E. So, here the inherited attribute which is the symbol table of E one is actually, you know, sent to E two.

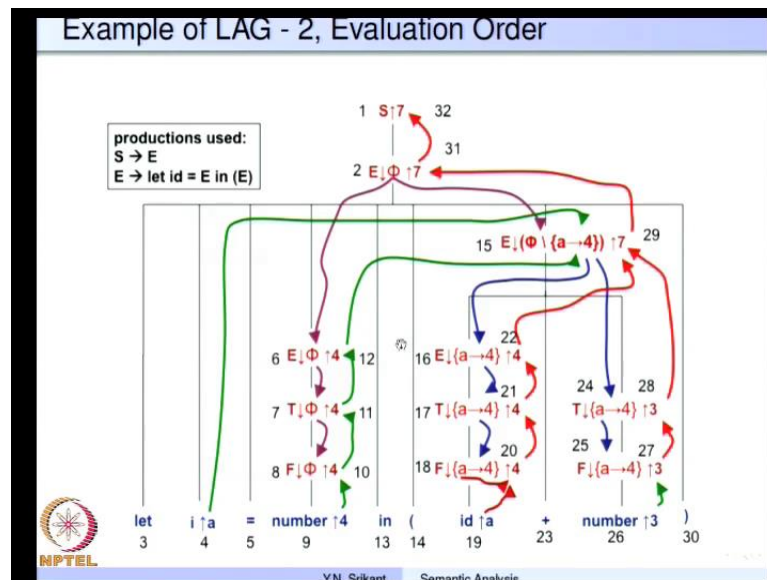
So, this is a perfectly valid computation; because it depends on; the, you know, E two dot syntab depends on the parent's syntab. So, nothing wrong with that. And then, the sym table of E three now is the symbol table of E one overridden with the association of id to E 2. So since id, E two are all, you know to the left of E three and the E one dot syntab is an inherited attribute. There is nothing wrong in this computation rule as well. So, this

is also satisfactory as far as the LAG property is concerned. None of the others, you know actually violate the properties of an LAG. And therefore, the grammar is indeed LAG.

Now, let us look at the modification of the let id equal to E two in E three rule. Modify it as return E three with id equal to E two. Let us say the semantics of this statement is the same as this let id equal to E two in E three. So, the semantics would be evaluate E three with the occurrences of id replaced by the value of E two. So, the attribute grammar actually does not change at all; because it is very similar to this rule. So, we are going to use the association of id to E two inside E three.

But, unfortunately when we do that the E three dot syntab now depends on the attributes of symbols to the right. So, it depends on the id name and it depends on E two dot val, which are to the right of E three. So, this is a clear violation of the LAG rules. Therefore, this grammar with the new production return E three with id equal to E two is not a L attributed grammar.

(Refer Slide Time: 09:19)



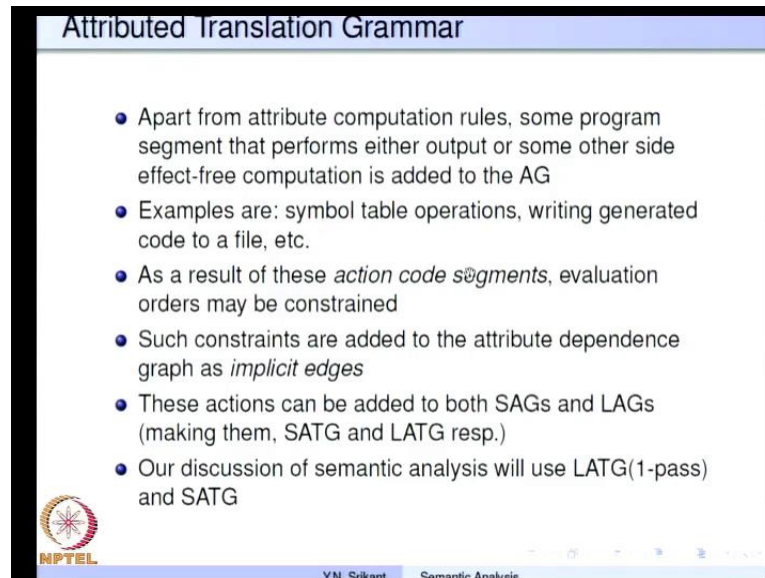
So, I promise to show the evaluation sequence using the dfVisit procedure. So, it is a same example of evaluating this expression. Let I equal to a plus four in a plus three. So, the visit sequence is written down here.

So, we start here and then we go to two, the symbol table is initialized to phi, then we actually visit three, then four and then five and then we visit six. So, this is the dfVisit strategy. So, this attribute is evaluated here; the inherited attribute symbol table, which is passed on to seven and then eight as well. At this point, the number four is passed upwards as a synthesized attribute. So, these are the visits; ten, eleven and twelve. So, once these visits are completed, we actually go to number thirteen. So, this is ten, eleven, twelve, etcetera.

And then once we do that, you know, the thirteen and fourteen are two terminal symbols here. So, they really do not have any semantic action associated with them. We go to fifteen. We are now ready to evaluate the symbol table here using the inherited attribute of the parent and sibling. So, that is done here. And then, it is passed on to. The visit continues; the symbol table is passed on to eighteen. And, here the value of a is looked up in the symbol table which produces four. This becomes a synthesized attribute, which is now passed on to twenty, twenty one and twenty two.

Finally, after twenty two we visit plus, which is twenty three. Now, this visits; twenty four and twenty five produce the number three, which is again passed upwards as synthesized attribute. And here, the value four from this and the value three from this are added to produce seven, which is again passed as an synthesized attribute to this visit number; during visit number thirty one. And then, finally the value is produced in S in visit number thirty two. So, this is the dfVisit sequence that is used to evaluate the attributes. And if you look at it, you know, the values always flow from left to right. They really never flow from right to left or they flow from top to bottom or bottom to top, but never from right to left.

(Refer Slide Time: 12:10)



The slide is titled "Attributed Translation Grammar" and contains a list of seven bullet points. In the bottom left corner, there is a circular logo for NPTEL (National Programme on Technology Enhanced Learning) with the text "NPTEL" below it. In the bottom right corner, there is a small navigation bar with the text "Y.N. Srikant" and "Semantic Analysis".

- Apart from attribute computation rules, some program segment that performs either output or some other side effect-free computation is added to the AG
- Examples are: symbol table operations, writing generated code to a file, etc.
- As a result of these *action code segments*, evaluation orders may be constrained
- Such constraints are added to the attribute dependence graph as *implicit edges*
- These actions can be added to both SAGs and LAGs (making them, SATG and LATG resp.)
- Our discussion of semantic analysis will use LATG(1-pass) and SATG

Now, let us look at a classification of, you know, attribute grammars into pure attribute grammars and attributed translation grammars. So, in pure attribute grammars; that is what we have studied so far. We asserted that the attribute computation rules do not have any side effects. So now, suppose you know we permit the attribute among the attribute computation rules, some small program segments like procedure calls or function calls that perform either output or some other side effect free operations; and we add this to the attribute grammar. Then, this entire system is called as an attributed translation grammar.

So, what are the actions possible? What are the program segments capable of doing without any serious side effects? For example, they can be symbol table operations; insertion of something into the symbol table or deletion of something from the symbol table, etcetera, writing generated code to a file. These are very important, you know, operations that can be performed.

So, when we say side effect free computation, none of the contents, of you know, denotes the other attributes, etcetera are modified by such operations. That is what we really want. So, such side effect free computations when added produce an attributed translation grammar. So, as a result of these action code segments, the evaluation orders may become a bit constrained, restricted. All the orders are not possible. I will show you an example of this very soon.



So, such constraints are added to the attribute dependence graph as implicit edges. And, these actions can be added to both the S attributed grammars and L attributed grammars. So, in that case they become SATG and LATG respectively. So, in our semantic analysis we are going to use LATG s of 1-pass variety. And, of course SATG s which are always 1-pass.

(Refer Slide Time: 14:44)

```

%%
lines: lines expr '\n' {printf("%g\n", $2);}
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr {$$ = $1 + $3;}
/*Same as: expr(1).val = expr(2).val+expr(3).val */
      | expr '-' expr {$$ = $1 - $3;}
      | expr '*' expr {$$ = $1 * $3;}
      | expr '/' expr {$$ = $1 / $3;}
      | '(' expr ')' {$$ = $2;}
      | NUMBER /* type double */
      ;
%%

```

The slide also features the NPTEL logo and the text 'Y.N. Srikant Semantic Analysis' at the bottom.

So, here is an SATG for desk calculators. So, it appears very familiar because this is nothing but a YACC specification for the desk calculator. So, let us look at the most important production expression going to expression plus expression, expression minus expression, expression star expression and expression slash expression, then parenthesized expression and number. So because we have only synthesized attributes, you know the computation rules can be placed at the end of the production. For LATG s, this is not the way to do it. It is different. We are going to see it very soon.

So, dollar dollar equal to dollar one plus dollar three. It computes the value of the left hand side expression, in terms of the values of the two right hand side expressions. So, dollar one corresponds to the first expression and dollar three corresponds to the second expression. The reason why it has become dollar three instead of dollar two is the symbols are numbered as one, two, three on the right hand side. So, this is number one, this is number two and this is number three.

So, again this is the YACC syntax for writing the attribute computations. Minus star and slash are very similar. For parenthesized expressions, there is nothing to do. It is just a copy of the expression. And, for number also when we do not write any rule, YACC assumes that there is a copy operation. So whatever is the attribute of number, it is copied to the expression. So, that is a default rule. So, now this particular computation had just one side effect, free state action; that is, the print the value. That is all. It did not have any other action. Suppose we permit, you know variable names to be introduced into the desk calculator.

So, now we have extra productions; name equal to expression. And, of course name itself. The rest of the productions are the same. So when we use name, the value of the name is actually obtained from the name table. And, whenever we say name equal to expression, the value of the name along with the name is introduced into the name table. So, here is the action to do that. The production in expression going to name equal to expression. So, symlook is a procedure which looks up a symbol table. And, the parameter is the name; that is dollar one.

So if the name is present in the symbol table, it produces a pointer pointing to that entry in the symbol table. And, if the name is not there in the symbol table, then it introduces that name into the symbol table. And then, you know returns the pointer to the new entry in S p. So once that happens, we can insert the value into the symbol table corresponding to that entry of S p. So, S p pointer value equal to dollar three; introduces the value of expression and associates it with S p dot, you know with the name which is pointed to the S p. And, the value returned by this entire production is the value of the third symbol; that is, expression. So, dollar dollar equal to dollar three does that.

And when we use a name, we look up that name. So, in the symlook dollar one looks up that name in the symbol table and returns a pointer to it. If the name is not present in the symbol table, it is introduced into the symbol table and a default value of 0 is initialized into the value field of the name. And, dollar dollar happens to be S p pointer value. So, the value of that entry is returned as the value of the expression on the left hand side. So, these are really actions which do not modify any other attribute of the grammar. But, they are certainly are not attribute computations. So, that is why this is a, you know, synthesized attribute translation grammar; SATG.

(Refer Slide Time: 19:29)


**Example 3: LAG, LATG, and SATG**

LAG (notice the changed grammar)

1.  $Decl \rightarrow DList\$$
2.  $DList \rightarrow D D'$
3.  $D' \rightarrow \epsilon \mid ; DList$
4.  $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
5.  $T \rightarrow int \{T.type \uparrow := integer\}$
6.  $T \rightarrow float \{T.type \uparrow := real\}$
7.  $L \rightarrow ID L' \{ID.type \downarrow := L.type \downarrow; L'.type \downarrow := L.type \downarrow;\}$
8.  $L' \rightarrow \epsilon \mid . L \{L.type \downarrow := L'.type \downarrow;\}$
9.  $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

LATG (notice the changed grammar)

1.  $Decl \rightarrow DList\$$
2.  $DList \rightarrow D D'$
3.  $D' \rightarrow \epsilon \mid ; DList$
4.  $D \rightarrow T \{L.type \downarrow := T.type \uparrow\} L$
5.  $T \rightarrow int \{T.type \uparrow := integer\}$
6.  $T \rightarrow float \{T.type \uparrow := real\}$
7.  $L \rightarrow id \{insert\_symtab(id.name \uparrow, L.type \downarrow); L'.type \downarrow := L.type \downarrow;\} L'$
8.  $L' \rightarrow \epsilon \mid . \{L.type \downarrow := L'.type \downarrow;\} L$



Y.N. Srikant Semantic Analysis

So this, now we go on and continue with another example. This example actually provides a changed grammar for the declarations that we saw before. So, why should we change the grammar? We change it because the previous grammar was not L L one. And for specifically LATG, we require a grammar which can be passed either by L L parsers or by recursive descent parsers.

So, here the change is in removing the left recursion. So, declaration going to DList dollar; DList going to D D prime; D prime going to epsilon or semicolon DList; rest of it is the same. Here also for the list of identifiers, we remove the left recursion and make it a right recursion. So, L going to I D L prime and L prime going to epsilon or comma L. Rest of the productions are the same. And, the semantic actions in the case of LAG can always be return at the end of the production; simply because the order in which we write the semantic rules within the production is not very relevant. You know, the dependences actually indicate the order in which the computations must happen.

So, given an attribute grammar of this kind, it will not be... this does not have any actions associated with it. So, and we will not be in a position to translate this into a program directly; whereas if you consider the LATG, we can translate this entire program into a, sorry, entire grammar into a program using an automatic generator. So, let us see how the LATG s specification is different from the LAG s and SAG s specification. The grammar here is the same that we had here.

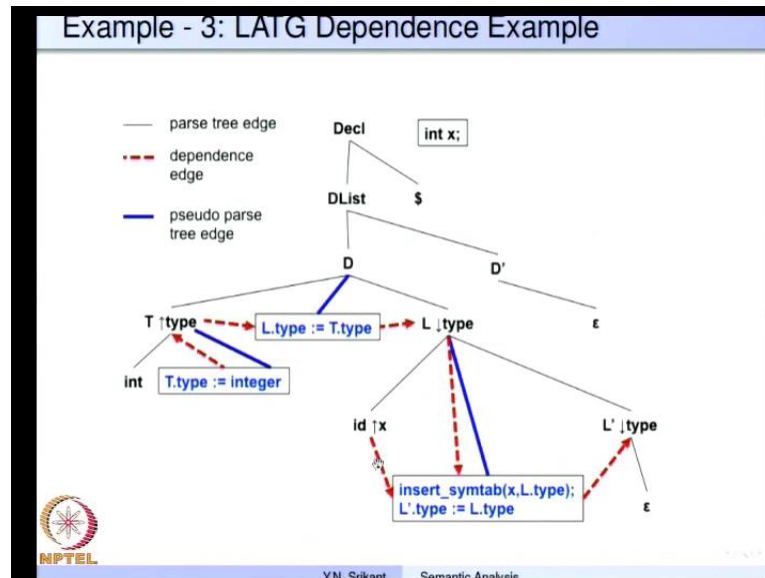
So, for the productions declaration going to DList dollar, D list going to D D prime and D prime going to epsilon or semicolon DList, there are no actions. So, let us consider this number four. The rule of computation can be attached at the end, but this is LATG. Therefore, initialization of the inherited attributes of L has to be done just before we actually parse the string generated by this L. So, L dot type equal to T dot type. So, T dot type is a synthesized attribute of T; L dot type is the inherited attribute of L. So, we are computing that just before L. So, this is the characteristic of LATG. compute the inherited attributes of a non-terminal, just before that non-terminal is processed.

Then we have T to int, which the rule can be attached at the end. It really, you know, does not matter. So, T to float; there is no other order possible; T dot type equal to real. Now, for the production L going to id and L going to epsilon or semicolon L, we have extra work to do. For the production L going to id, we have insert symbol table. So, the routine is called with the name of the identifier and the type of the identifier.

So, the L dot type is the inherited attribute of the left hand side non-terminal. It is already available. So, we pass id dot name and L dot type. So, this is inserted into the symbol table and the associated, you know, the appropriate fields are filled. And after that, the attributes of L prime are computed just before L prime. So, L prime dot type inherited attribute equal to L dot type; again, from the left hand side of the production. So remember, we can compute the inherited attributes of the symbol just before it. But, the synthesized attributes are available from below. So, there is no question of computing the synthesized attributes of L prime in and providing rules for it in this production.

So, L prime going to epsilon has no rules associated with it because there is no computation necessary. Actually, it ends a declaration list. So, there is nothing to do. Whereas if you have a semicolon L, then, sorry, comma L; after the comma, we initialize the inherited attribute of L. So, L dot type equal to L prime dot type. And then, processes the string generated by L. So, the attribute computation rules are actually interspersed with the production, among the production symbols. And, the order in which we execute these attribute computation rules is now constrained.

(Refer Slide Time: 24:54)



So, let me show you what happens in this case. Here is a very simple sentence; int x semicolon. And, here is a parse tree for it. So, declaration going to DList dollar and DList going to D D prime, D prime goes to epsilon. So, here this D goes to T L and then T goes to int, L goes to you know id, and then a comma, which is, you know which is not in the picture because it is not very important. And then, L prime. And, L prime goes to epsilon.

So, here if you observe the boxes, these are the attribute computation rules and the action code actually. So, insert symtab is the action code, whereas the others, L dot type equal to T dot type etcetera are the attribute computation rules. So, what we really have done is to introduce a pseudo edge and a pseudo node corresponding to the attribute computation rule. And then, you know the order in which the rules have to be; attribute computation rules have to be executed is shown in red.

So, if you look at this position, it has this L dot type equal to T dot type has been inserted between T and L. So, the production is D going to T L and the node is inserted right here. Let us look at the grammar to check whether it is correct or wrong. It is indeed correct. So, D going to T, then the computation rule and followed by L. So, if we simply do the same dfVisit on this augmented parse tree, so for example, even for this, the attribute computation rule is between id and L prime; L going to id semicolon, you know, L prime. So, let us see that here.

For example, so id and L prime. So this part, you know we have a semantic action here. right. So, there is no semicolon. Semicolon is here. So, id and L prime and the semantic action is in between. This semantic action has been made into a pseudo node and a pseudo edge is attached to its parent L. So, if we consider this augmented parse tree and simply conduct a dfVisit and execute the attribute computation rules in that order, automatically the attribute computations happen properly. So we can see that; you know declaration, then DList and then D, then T, then int. So, this attribute gets computed now when we return from int.

And once we do that, we go up again to D and then come down to this. Right. Sorry. So after that, T and then int and then we need to visit T dot type equal to integer. So, we compute the attribute of T and that produces type information here and that would be int. And, now we go up to T and then come down to this action. So, we do L dot type equal to T dot type; that gets the attribute of L. Ready for the next visit. Then, we go up again and then come down to L. So, this attribute is already ready. Now, we go to id and then we go up again and then we come down to this action.

Now, we can see that x and L dot type are both ready. So, this semantic action of inserting into the symbol table can be carried out properly. Then we initialized L prime dot type equal to L dot type. So, this attribute gets ready now. We go back and then come down to L prime. Finally, you know this epsilon is visited. We go up back all the way, then visit D prime epsilon, then dollar. And, that is the end of the visit.


So in the case of L a T G, the parse tree is first constructed without the actions. And, once we construct the parse tree looking at the productions, we insert dummy nodes for the action segments; attribute computation rules and action segments. And once that is ready, we do a dfVisit on the parse tree. Execute the semantic actions as necessary and that gets the attributes evaluated.

(Refer Slide Time: 29:49)

Example 3: LAG, LAIG, and SATG (contd.)

SATG

1.  $Decl \rightarrow DList\$$
2.  $DList \rightarrow D \mid DList ; D$
3.  $D \rightarrow T L \{ patchtype(T.type \uparrow, L.namelist \uparrow); \}$
4.  $T \rightarrow int \{ T.type \uparrow := integer \}$
5.  $T \rightarrow float \{ T.type \uparrow := real \}$
6.  $L \rightarrow id \{ sp = insert\_symtab(id.name \uparrow); L.namelist \uparrow = makelist(sp); \}$
7.  $L_1 \rightarrow L_2 , id \{ sp = insert\_symtab(id.name \uparrow); L_1.namelist \uparrow = append(L_2.namelist \uparrow, sp); \}$



Y.N. Srikant Semantic Analysis

So, the SATG for the same language of declarations is also shown here for comparison. So, D going to T L is the same. So, the production is the same. And we just, you know, call a function; this thing production, a procedure called patch type. I will tell you why that is needed after we go down a little bit. So T to int, the rule of computation is very simple; T dot type equal to integer. T to float is T dot type equal to real. Now, we have L to id and then we have L to L comma id. Since, this is bottom up parsing there is no problem with left recursion here.

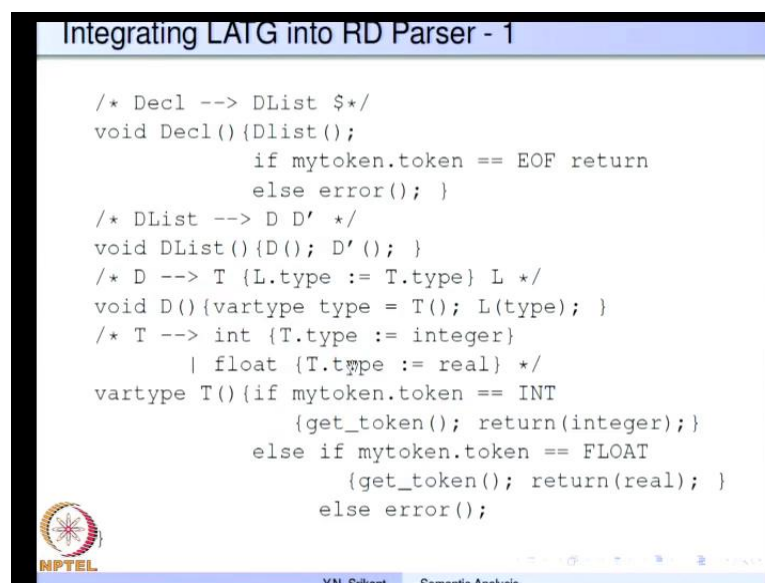
For L to id, we insert the name into the symbol table and then we create a list of names in L dot name list. And, that carries this id as well. Why should we do this? The same is done here as well in L two comma id. We insert the name into the symbol table and append the new name to the L two dot name list and that is sent out as L one dot name list. The problem is when we have come up to this point; we have parsed the type information in T. And, that about attribute is completed as T as well. We have parsed the name list which is L. And, if we do not carry the list of names in L as an attribute of L, how do we actually attach the type information to each of the names which L produces? That is the question.

So, now we have a list of names available as a synthesized attribute. Here, the type information is available as a synthesized attribute of T. So, we can execute an action; patch type T dot type comma L dot name list. So, which traverses the name list and since

these are all entries in the symbol table, it enters the type information for each name into the symbol table and goes to the next name. So, in this manner the symbol table can be constructed for this declaration list using SATG s.

So, SATG s can be used with the YACC and I showed you a several examples already. So, these are all, you know, translated to C code automatically and the YACC specification. Automatically, this becomes a program which can be; rather, this becomes an attribute grammar which can be used for processing by YACC.

(Refer Slide Time: 32:43)



```
Integrating LATG into RD Parser - 1

/* Decl --> DList $*/
void Decl(){Dlist();
            if mytoken.token == EOF return
            else error(); }

/* DList --> D D' */
void DList(){D(); D'(); }
/* D --> T {L.type := T.type} L */
void D(){vartype type = T(); L(type); }
/* T --> int {T.type := integer}
   | float {T.type := real} */
vartype T(){if mytoken.token == INT
            {get_token(); return(integer);}
            else if mytoken.token == FLOAT
            {get_token(); return(real); }
            else error(); }
```

NPTEL  
Y.N. Srikant - Semantic Analysis

So far, we just saw the grammar. You know, and then I told you that it is possible to evaluate the attributes over the parse tree. It is also possible to integrate the LATG into a recursive descent parser. So that, parsing and attribute evaluation happen in a simultaneous manner. Just like the attribute computations can happen hand in hand along with L R parsing.

So, the same can be done in recursive descent parsing as well. So, let us look at the recursive descent parser and the attribute computations for the same LATG that we saw a few minutes ago. First production is declaration going to DList dollar. So, the function is void declaration, the body is call DList and then the next one is dollar. That is the end of file. So, if mytoken dot token equal to E O F, then return. Otherwise, obviously it is an error. So, this is the function for declaration non-terminal.



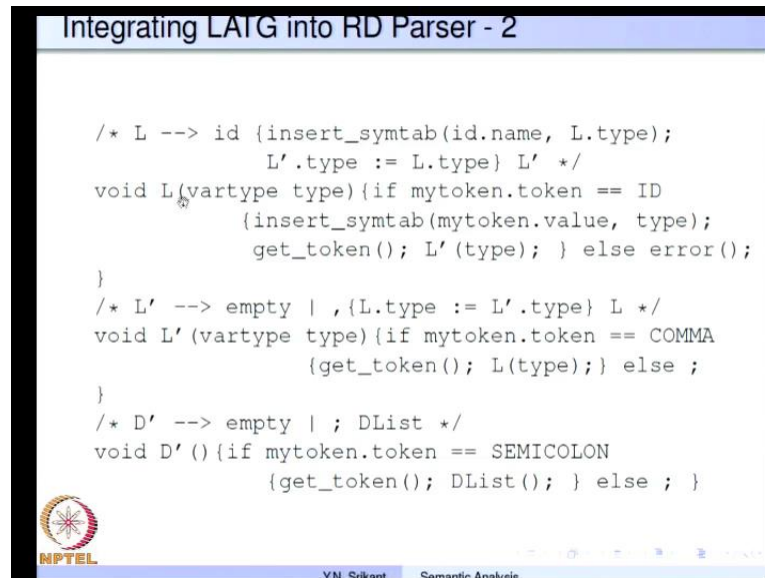
The next production is DList going to DD prime. So, again the function is called DList. It returns nothing. The body consists of a call to D and another call to D prime. That is it. The third production is D going to T L and there is a semantic action in between L dot type equal to T dot type. So, the function produced is void D. It is not returning any synthesized attribute. So, that is why it is void. So, vartype type is equal to a call to T. So, T produces a synthesized attribute and that is actually returned into the variable type. So now, we pass that type information into L as a parameter, which is an inherited attribute. So, here L dot type is inherited and T dot type is synthesized.

So, T is returning a value and that is given to the variable through this assignment. L dot type is an inherited attribute and that becomes an incoming parameter to the function L. So, that is how this production is structured in recursive descent parsing.

So, consider T going to int or T going to float. So, this is quite straight forward; vartype T. So, if mytoken is int, then get token and return integer. So, this integer is the type information. And, remember T is returning a type information called vartype. So, see this here. right. And, that is the int part. If the token happened to be float, then we get the next token and return real.

So vartype, you know also has a real unit. So, real value is; return type is returned as the type of T; otherwise, error. So, when we have a synthesized attribute which is going out, it is actually the result of the function, result type of the function. And, whenever there is an inherited attribute coming into a non-terminal, a corresponding function will really have an incoming attribute.

(Refer Slide Time: 36:33)



```
/* L --> id {insert_syntab(id.name, L.type);
    L'.type := L.type} L' */
void L(vartype type){if mytoken.token == ID
    {insert_syntab(mytoken.value, type);
    get_token(); L'(type); } else error();
}
/* L' --> empty | , {L.type := L'.type} L */
void L'(vartype type){if mytoken.token == COMMA
    {get_token(); L(type);} else ;
}
/* D' --> empty | ; DList */
void D'(){if mytoken.token == SEMICOLON
    {get_token(); DList(); } else ; }
```

So, we will now see L. So, that will make it very clear. So, L going to id and then there is an action and finally L prime. So, this is the production. So, L does not return any result. It only takes an inherited attribute.

So, there is a parameter corresponding to it; vartype type. And then, the body simply says if mytoken dot token equal to id, so that is the parsing part. Now, the action is introduced into the recursive descent parser; insert symbol table. So, id dot name is nothing but mytoken dot value and L dot type is nothing but the incoming parameter type. So, this function is executed. Then, we get the next token and call L prime. So, L prime dot type equal to L dot type is the initialization just before L. So, that code gets, you know, executed; because this type information is already available as the incoming parameter. There is no need for another computation here. So, that is automatically available and used here; otherwise, error.

The next production is L prime going to empty or L prime going to comma L. So, after the comma there is an attribute computation. So, the function becomes void L prime. So, there is no return of any synthesized attribute from here. The incoming attribute, incoming parameter is the inherited attribute; vartype type. So, if mytoken dot token is comma; so that is, this production is applicable, then get token and call L with the inherited attribute type. So, type is available as an incoming parameter here. So, else; so

if the token is not comma, then the production applied is L prime going to empty. So, we have a null statement semicolon here.

The last statement is D prime going to empty or rather last production; D prime going to semicolon DList. There are no, you know, attribute computations here. So, it is just a parser part that is present in the recursive descent parser function D prime. So, if my token dot token equal to semicolon, then get token call DList. Otherwise, it is empty part; so a null statement semicolon.

So, this is how the recursive descent parser embeds, you know, the semantic actions of an LATG. The most important part to remember here is the inherited attributes are parsed as incoming parameters to a function and corresponding to the non-terminal. And, the synthesized attributes are the outgoing results of the function corresponding to that non-terminal.

(Refer Slide Time: 39:42)

```
Example 4: SATG with Scoped Names

1. S --> E { S.val := E.val }
2. E --> E + T { E(1).val := E(2).val + T.val }
3. E --> T { E.val := T.val }
/* The 3 productions below are broken parts
   of the prod.: E --> let id = E in (E) */
4. E --> L B { E.val := B.val; }
5. L --> let id = E { //scope initialized to 0;
                    scope++; insert (id.name, scope, E.val) }
6. B --> in (E) { B.val := E.val;
                delete_entries (scope); scope--; }
7. T --> T * F { T(1).val := T(2).val * F.val }
8. T --> F { T.val := F.val }
9. F --> (E) { F.val := E.val }
10. F --> number { F.val := number.val }
11. F --> id { F.val := getval (id.name, scope) }
```

So now, let us look at the SATG version of the expression evaluation grammar. So, it is the same; you know, grammar with the special production E going to let id equal to E in E. Let us see how the SATG is written for such a grammar. There is a minor problem with the grammar and it has to be rewritten. So, let us go through this in a production by production fashion. So, S to E does not post problems. So, the semantic evaluation is straight forward; S dot val equal to E dot val. E to E plus T, obviously does not post any

problems. So,  $E \text{ one dot val} = E \text{ two dot val} + T \text{ dot val}$ .  $E$  to  $T$  is just a copy. So, there is nothing wrong here as well.

Now, when we come to the production, you know,  $E$  going to let  $id$  equal to  $E$  in  $E$ , we really cannot process it as it is. The reason is we are using bottom up parsing strategy for the SATG. So, if we attach a semantic action at the end of the production, that is at this point, then we would have already parsed the second; this first  $E$  and the second  $E$ . And, the association of  $id$  to  $E$  cannot be made available to the second  $E$  at all.

Therefore, somehow we must attach an action in the middle somewhere to introduce the association of  $id$  to  $E$  into a symbol table. And the symbol table being a global entity, this  $E$  will also know about it. So to do that, we are going to break this production;  $E$  going to let  $id$  equal to  $E$  in  $E$  into several productions.

The first production would be  $E$  to  $LB$ . So, the second; the first part  $L$  produces let  $id$  equal to  $E$  and the second part  $B$  produces in  $E$ . So at the highest level,  $E \text{ dot val}$  is nothing but  $B \text{ dot val}$ . So, that is a fairly simple attribute computation, but when we come to the production  $L$  going to let  $id$  equal to  $E$ . So, there are several points to be noted here. The first is we are going to use the nesting level as the scope of a particular name. So, the scope is first initialized to zero. So when, you know, there is no other nesting possible, the scope value is 0. And, whenever we have a new association let  $id$  equal to  $E$ , we are going to increment the scope. So, scope first initialized to 0. This is the comment.

Now, as soon as we parse let  $id$  equal to  $E$ , we know that there is a new scope generated. So, we increment the scope, insert the name  $id \text{ dot name}$  into the symbol table with the new scope and the value that is to be associated with  $id \text{ dot name}$  is  $E \text{ dot val}$  at that particular level or scope. So now, scope is a global variable. It is being manipulated in this production. So, this is the reason it becomes an SATG. In the production  $B$  going to in  $E$ ,  $B \text{ dot val}$  is obviously  $E \text{ dot val}$ . So, that is not an issue at all. And now after this, you know,  $E$  now has available the scope symbol table. So, we will see the use of this symbol table somewhere down here in the production  $F$  going to  $id$ .

But at this point of time, let us assume that  $E$  has been parsed and evaluated. So,  $E$  has produced a value using the association of  $id$  to  $E$ . So,  $B \text{ dot val}$  is  $E \text{ dot val}$ . Now that, we have actually exited this scope; so let  $id$  equal to  $E$  in  $E$ . So, this is the end of the scope

for the name *id*; name associated with *id*. So, we must do two things; the name that was introduced at that scope has to be deleted and the scope value has to be reduced by one. So, we do both these. So, we do both these and delete entries with the scope as parameter removes all the entries from the symbol table with this scope. And, scope minus minus reduces the scope by one. So, we are back to the enclosing nesting level. And, thereby we are free to introduce any other entries at new levels from now on.

$T \rightarrow T * F$  is very simple. So,  $T \rightarrow T \cdot val$  equal to  $T \rightarrow T \cdot val + F \cdot val$ . Similarly,  $T \rightarrow F$  and  $F \rightarrow E$  and  $F \rightarrow number$ . So when we go to  $F \rightarrow id$ , the value of  $F$  is obtained by looking at this symbol table with the present scope. Scope being a global symbol, whatever value it holds is the scope at which we entered  $id$  equal to  $E$ . So, then the same name can be obtained using the scope entry. And, that is returned as synthesized attribute of  $F$ .


So, this is how the same, you know, expression grammar with LAG. It can be modified into SAG. But, this breaking of production is essential here. This is actually a very basic principle as we will see when we do semantic analysis of “if then else statements” and “while do” statements. We require a similar breaking up of productions to make sure that semantic checks for the expressions are inserted at appropriate points.

(Refer Slide Time: 46:39)

LAI G for Sem. Analysis of Variable Declarations - 1

- 1  $Decl \rightarrow DList\$$
- 2  $DList \rightarrow D \mid D ; DList$
- 3  $D \rightarrow T L$
- 4  $T \rightarrow int \mid float$
- 5  $L \rightarrow ID\_ARR \mid \_D\_ARR . L$
- 6  $ID\_ARR \rightarrow id \mid id [ DIMLIST ] \mid id BR\_DIMLIST$
- 7  $DIMLIST \rightarrow num \mid num . DIMLIST$
- 8  $BR\_DIMLIST \rightarrow [ num ] \mid [ num ] BR\_DIMLIST$

Note: array declarations have two possibilities  
`int a[10,20,30]; float b[25][35];`


VN. Srikant    Semantic Analysis

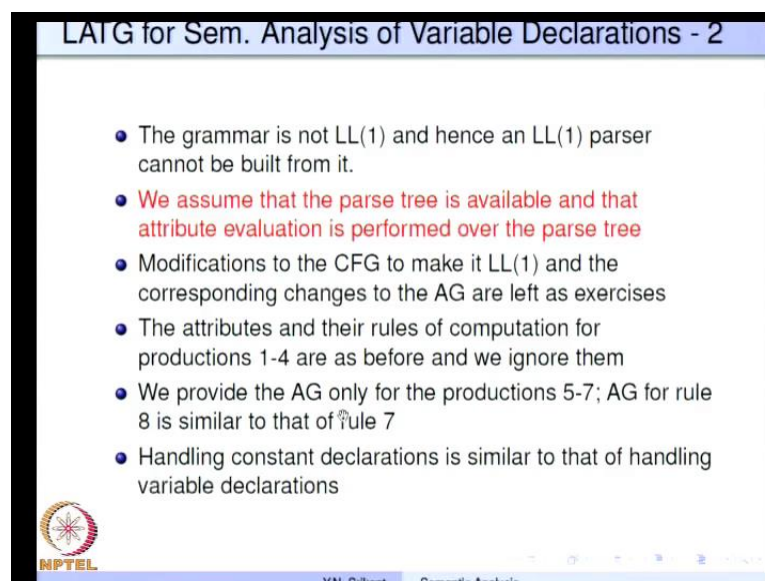
Now, let us move on. Let us look at semantic analysis of declarations. We saw the declarations in a slightly diluted fashion so far. So in other words, the only thing we saw

was a simple declaration of the form  $D$  going to  $T L$ ; where  $T$  is a type and  $L$  is a list of names. But, suppose we add arrays into it and to make it more interesting, we also permit arrays to be declared in more than one way.

For example, arrays can be declared as `int a with 10, 20, 30` as the three ranges for the three dimensions. Or, we may permit the arrays to be declared as separately separated by these square brackets; `int a[25, 35]`. So, both these varieties are permitted in this grammar. Let us see how it does it. So,  $L$  is a list of names or simple names or array names. So,  $L$  is either `id array` or `id array comma L`. So, in other words it is either a single name or a list of names. `id array` is either a simple `id` or `id` followed by the dimension list. That is something like this. Dimension list produces `number comma DIMLIST`. So, it produces a list of these dimensions.

So, this type of declaration is taken care of `id bracket DIMLIST bracket`. And, `id BR_DIMLIST`; so `BR_DIMLIST` says look at, you know, have one bracket, then a number followed by another bracket. Or, a similar structure of `bracket num bracket` followed by `bracket BR_DIMLIST`. So, the declarations of this type are taken care of by this production. And, declarations of this type are taken care of by such productions. So, let us see how to write LATG s for such declarations.

(Refer Slide Time: 49:00)



LATG for Sem. Analysis of Variable Declarations - 2

- The grammar is not LL(1) and hence an LL(1) parser cannot be built from it.
- We assume that the parse tree is available and that attribute evaluation is performed over the parse tree
- Modifications to the CFG to make it LL(1) and the corresponding changes to the AG are left as exercises
- The attributes and their rules of computation for productions 1-4 are as before and we ignore them
- We provide the AG only for the productions 5-7; AG for rule 8 is similar to that of rule 7
- Handling constant declarations is similar to that of handling variable declarations

NPTEL

V.N. Srikant Semantic Analysis

So, there are a couple of points that we need to note here. The grammar that I presented here, this is obviously not LL one. So, it is very easy to see. You know, number is

common between these two; id is common between these three; id arrays are common between these and so on. And, num bracket num is common between these two. So, all those require some factoring and so on. So, to make it LL one. So, we assume that the parse tree is available and that attribute evaluation is passed over the parse tree by augmenting it with dummy symbols for actions and so on and so forth.

So, modifications to the CFG to make it really LL one. I showed you a grammar which can be made into a recursive descent parser. So, that is an example of an LL one grammar along with its semantic actions. So, making this grammar into LL one and changing the semantic actions appropriately are left as exercises. Now, attributes and their rules of computation for the productions one to four are as before and we ignore them.

So up to this point, the grammar has not changed. So, I am not going to repeat the semantic actions for these four. They are as before. So, we provide the attribute grammar only for the productions five to seven. The attribute grammar for the eight is very similar to that of rule seven. So, eight is this type of declaration. So, that is here. Processing this is very similar to the processing that we do for six and seven. There is absolutely nothing different. So, we will do only for six and seven and leave eight for the exercises.

Finally, handling constant declarations is similar to that of handling variable declarations. So, variable declarations have a type attached to it. Whereas constant declarations, let us say we have `int a equal to 5`. So, we can treat this, you know as a variable a initialized to value five. So, the initial value can be stored along with the variable in the symbol table and use later for code generation.

Of course, some languages also have constant declarations themselves like in Pascal and C plus plus and so on. Constant declarations are available. So in such cases, the identifier or the name associated with the constant declaration is also entered into the symbol table just like a variable. But, a flag indicates that it is a constant. But, otherwise processing constant declaration is not very different from that of handling variables.

(Refer Slide Time: 52:08)

Identifier type information record

name	type	eletype	dimlist_ptr
------	------	---------	-------------

1. `type`: (simple, array)
2. `type` = simple for non-array names
3. The fields `eletype` and `dimlist_ptr` are relevant only for arrays. In that case, `type` = array
4. `eletype`: (integer, real, error type), is the type of a simple id or the type of the array element
5. `dimlist_ptr` points to a list of ranges of the dimensions of an array. C-type array declarations are assumed  
Ex. `float my_array[5][12][15]`  
`dimlist_ptr` points to the list (5,12,15), and the total number elements in the array is  $5 \times 12 \times 15 = 900$ , which can be obtained by traversing this list and multiplying the elements.

NPTEL

Y.N. Srikant Semantic Analysis

Then, the other thing we must note is each identifier has several pieces of information attached to it. So, this is the identifier type information record and all this information must be available in the symbol table also, corresponding to this particular name. So, the name of the identifier is available, then the type of the identifier, then the element type of the identifier and a pointer to the various dimensions. In case, that identifier is an array.

So, let us see what these fields are. The type field can take two values; either simple or array. These are the user defined scalar names. So, type is simple for non-array names and type is array for array declarations and the fields eletype and dimlist pointer are relevant only for the arrays.

So, what does eletype do? It stores the value; either integer or real or error type. And, this is the type of a simple identifier or the type of an array element. So, for example, if you have `float, my_array 5,12,15`, float is the type of the array element. And, these are the various dimensions; dimlist pointer points to a list of ranges of the dimensions of an array. So, C type array declarations are assumed here. So, if you take this example of `float, my_array 5,12,15`; so float as I already told is the eletype and then you know the type of this entire name `my_array` is array. So, that is what is filled here. Name of the array is `my_array`.

And then, there are three dimensions here; first dimension, second dimension and third dimension. So, 5 is the number of elements in the first dimension, 12 is the number of



elements in the second dimension and 15 is the number of elements in the third dimension. So, what we really do is first make a list; 5,12,15. And, make dimlist pointer point to such a list. And, that is what is hanging here. So, this points to a list; 5, 12, 15 is for this particular example.

Why we have to do this? The point is when we actually do some code generation or when we do, for example, offset computation, we will need to find the size of the array or the size of this slice, etcetera. So, if for example, if the language permits assigning slices of arrays to other arrays or other slices, then we may require the size of a slice. So, if we say simply my list, my\_array three, then what is the size of a single element of this my\_array three slice? That would be actually 12 into 15.

Whereas if we consider another slice, my\_array of three comma four, so that would be a small array of; each element would be an array of size 15. Whereas if we consider the whole array, then the size is 5 into 12 into 15; that is, 900 elements. So, if depending on what we require, we may have to traverse this list and produce the number of elements of the appropriate slice. So, this is the reason why we require such elaborate information in the identifier type information record. So, we will stop here and continue with semantic analysis in the next lecture.

Thank you.