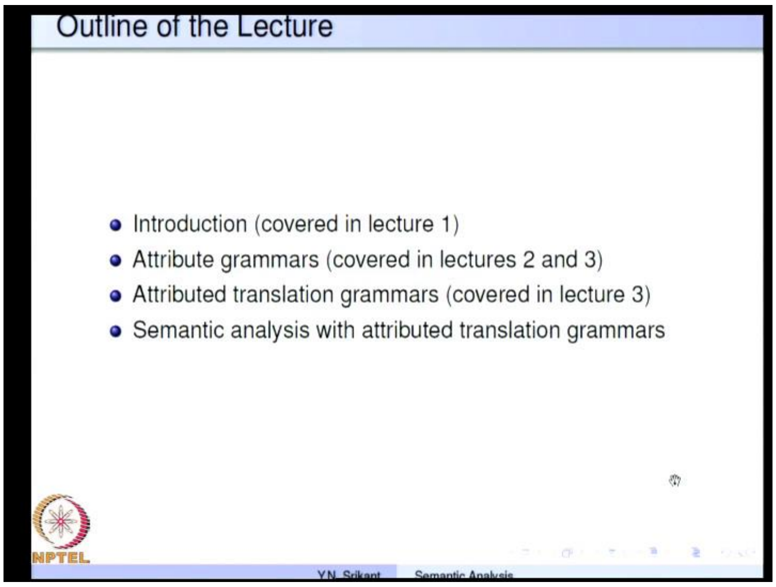
Principle of Complier Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 15
Semantic Analysis with Attribute Grammars Part – 4

(Refer Slide Time: 00:21)



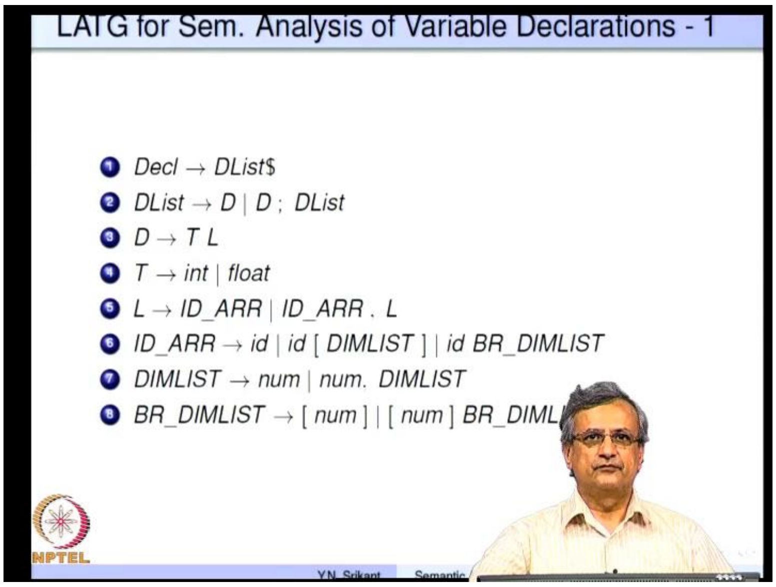
Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars (covered in lectures 2 and 3)
- Attributed translation grammars (covered in lecture 3)
- Semantic analysis with attributed translation grammars

Y.N. Srikant Semantic Analysis

Welcome to part 4 as the lecture on semantic analysis with attribute grammars. We have already looked at a few examples of attributes grammars and attributed translation grammars. Today we will continue with the semantic analysis part.

(Refer Slide Time: 00:35)



LATG for Sem. Analysis of Variable Declarations - 1

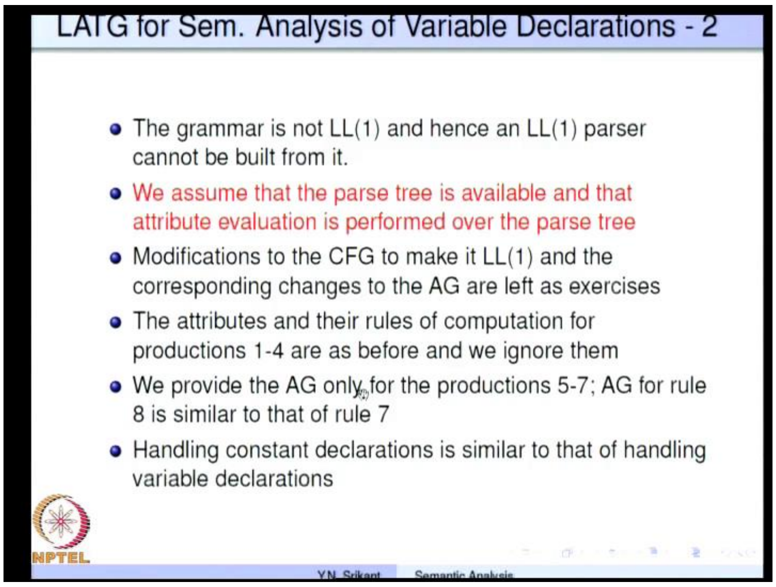
1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D \mid D\ ;\ DList$
3. $D \rightarrow T\ L$
4. $T \rightarrow int \mid float$
5. $L \rightarrow ID\_ARR \mid ID\_ARR\ ,\ L$
6. $ID\_ARR \rightarrow id \mid id\ [\ DIMLIST\ ] \mid id\ BR\_DIMLIST$
7. $DIMLIST \rightarrow num \mid num,\ DIMLIST$
8. $BR\_DIMLIST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

Y.N. Srikant Semantic

So, to do a brief recap, this was the grammar which I showed you last time and this is the grammar on which, we are going to do some semantic analysis. So, this is you know a short and grammar for declarations, similar you know to the decelerations in the language c. So, we have a D List which produces several declarations and then each deceleration of these of the form type followed by list of identifiers, we allow only two types; int and float. Then in the list of identifiers we allow either arrays or normal single identifiers.

So, arrays you know single identifiers would be just i d, whereas arrays would have two formats i d followed by DIMLIST in brackets, or i d followed by B R DIMLIST which produces again the several dimensions of the array each enclosed in a different set of brackets. Whereas, here only the dimensions will be you know listed here separated by a comma.

(Refer Slide Time: 01:55)



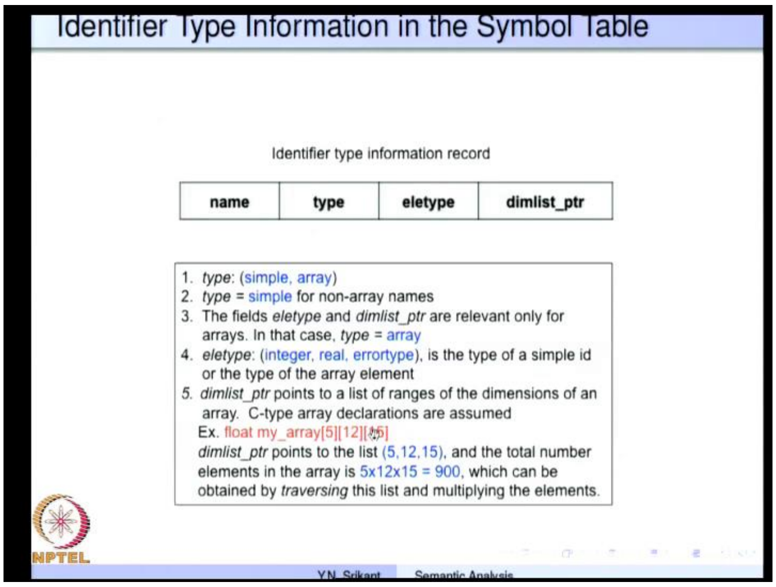
LATG for Sem. Analysis of Variable Declarations - 2

- 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

Y.N. Srikant Semantic Analysis

Here are a couple of important observations, first one is the grammar I showed you is not L L 1, so modifications are needed to make it L L 1. And therefore we assume that the parse tree is available and the attribute evaluation is performed over the parse tree, so we already know how to do this. The productions you know 1 to 4 we have already dealt with this in detail in the previous lectures, so we are not going to repeat that part here, we are going to concentration on these productions only. So, constant decelerations are similar to variable decelerations and we need to enter you know the identifier associated with the constant into the symbol table along with tag that it is a constant, but otherwise searching the symbol table etcetera is all very similar.

(Refer Slide Time: 02:58)



Identifier Type Information in the Symbol Table

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, errortype), 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

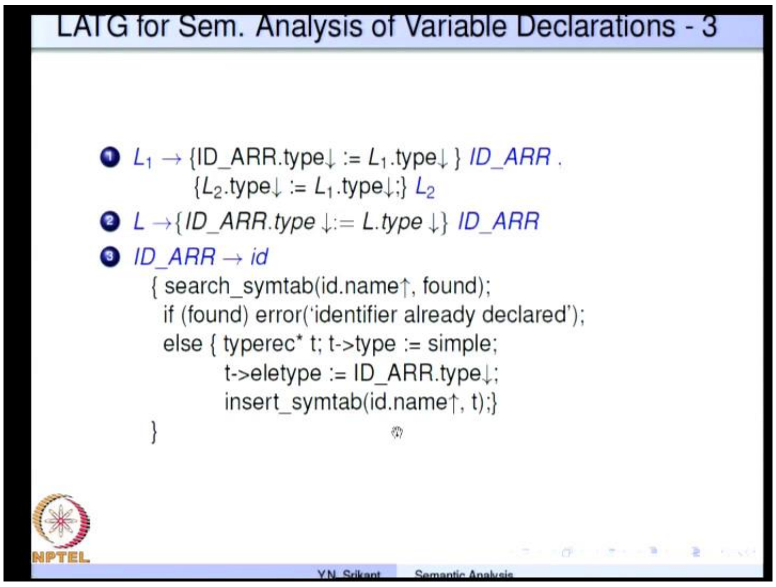
   Ex. float my_array[5][12][15]

   dimlist_ptr points to the list (5,12,15), and the total number elements in the array is 5x12x15 = 900, which can be obtained by *traversing* this list and multiplying the elements.

Y.N. Srikant Semantic Analysis

So, I showed you the identifier type information record last time. So, we have for each identifier we need to store its name, its type, the element type and you know a pointer to the list of dimensions. So, the type can be simple or array, whereas the element type is either integer or real, of course error type is used to take care of semantic analysis. So, the dimlist pointer points to a list of ranges of the dimensions of an array. For example, if we have float my array 5, 12, 15, then there is a list of items 5 comma 12 comma 15 and that is pointed to by dimlist pointer, this will be required later for you know to get the number of elements array, etcetera

(Refer Slide Time: 03:50)



LATG for Sem. Analysis of Variable Declarations - 3

```
1  $L_1 \rightarrow$ {ID_ARR.type↓ := $L_1$.type↓ } ID_ARR ,
        {$L_2$.type↓ := $L_1$.type↓;} $L_2$
2  $L \rightarrow$ {ID_ARR.type ↓:= L.type ↓} ID_ARR
3  ID_ARR $\rightarrow$ id
     { search_symtab(id.name↑, found);
       if (found) error('identifier already declared');
       else { typerec* t; t->type := simple;
              t->eletype := ID_ARR.type↓;
              insert_symtab(id.name↑, t);}
     }
```

Y.N. Srikant Semantic Analysis

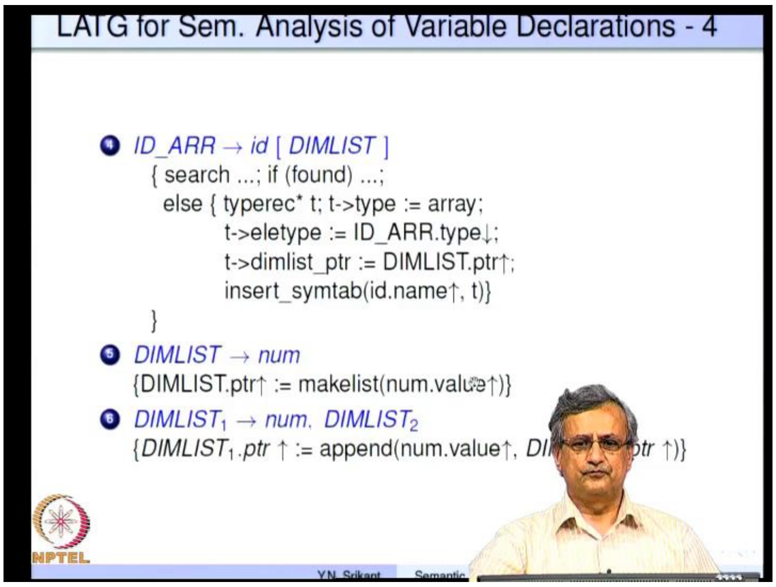
So, let us get started with production number 1, this is an L a t g and that is L attributed translation grammar, so therefore the actions have to be appear in built into the production itself. So, the actions before I D array, the non terminal I D array actually copy the inherited attribute L 1 dot type to inherited attribute I D array dot type. So, this is the initializations of the inherited attribute and then we have the I D array non terminal for which appropriate procedure would be called. Then we have initialization of the inherited attribute L 2 dot type with the value L 1 dot type and then the procedure L 2 for L 2 is called. So, this is quite straightforward, what we really do here is the take the type information and passes it on to all the elements in the list of identifiers produced by L. So, if we have L 2, I D array alone, then you know we just pass I D array dot type equal to L dot type.

Now, so we have you know the type information actually is synthesized in this you know T to int or D to float and the type information has to pass from this T into L, so this is how it works. So, let us remember that and then go further. So, now at this point L 1 dot type already has the type information, this was given to the you know the non terminal L by the production D going to T L, this is what we have seen before. So, I D array now produces i d, so this is the production that we have used, so what we do now? The semantic analysis is actually quite simple in this case, we search the symbol table, so such symtab is the function which does it, it takes the name of the identifier and a variable you know which is returned as found.

So, found is either true or falls depending on whether the name is found in the symbol table or not. If the name is already found; that means, the identifiers has already been declared, so there is an error message we just printed out by the compiler, identifier already declared. If the name is not found in the symbol table, then it is time to insert it into the symbol table.

So, what is the type of this particular name? It is actually I D array dot type, so we have to introduce the type information into the I D information recorded and then insert it to the symbol table. So, for that purpose we declare t as a temporary name, typerec star t, so t pointer type is set a simple, because we have a single identifier here and then t pointer eletype is made as I D array dot type, whatever is obtained from the above and insert some symtab insert this name with this particular type information into the symbol table.

(Refer Slide Time: 07:17)



LATG for Sem. Analysis of Variable Declarations - 4

```
● ID_ARR → id [ DIMLIST ]
    { search ...; if (found) ...;
      else { typerec* t; t->type := array;
             t->eletype := ID_ARR.type↓;
             t->dimlist_ptr := DIMLIST.ptr↑;
             insert_symtab(id.name↑, t)}
    }
● DIMLIST → num
    {DIMLIST.ptr↑ := makelist(num.value↑)}
● DIMLIST_1 → num, DIMLIST_2
    {DIMLIST_1.ptr ↑ := append(num.value↑, DI          ptr ↑)}
```

Y.N. Srikant   Semantic

So, what if I D array produces an array name? So, the name of the array is I D and DIMLIST is the list of dimensions of the array. So, the first part is very similar, you search the symbol table and if it already found, the name is already found, then an error message is issued. If the name is not found in the symbol table, we introduce it into the symbol table, so let us see how to fill up the information into the identifier record. So, we have the same typerec star t, a temporary t pointer type is now set as array, in the previous case it was set as simple, now it is set as array.

The element type is same as I D array dot type, which is obtained from the top, so if you have int my array 15, 20, 25, then int would be the type of the array element which is obtained from the top. Then t pointer, dimlist pointer is set as DIMLIST dot p t r. So, we will see how to synthesis DIMLIST dot p t r in the productions below. So, after this the symbol table call is made to insert the name with this type information into the symbol table. So, if you observe even though it is t g, in most of the cases the actions appear at the end, this is perfectly ok, because actions can appear in the middle or at the end in the case of l a t g depending on the requirement for computing the inherited attributes.

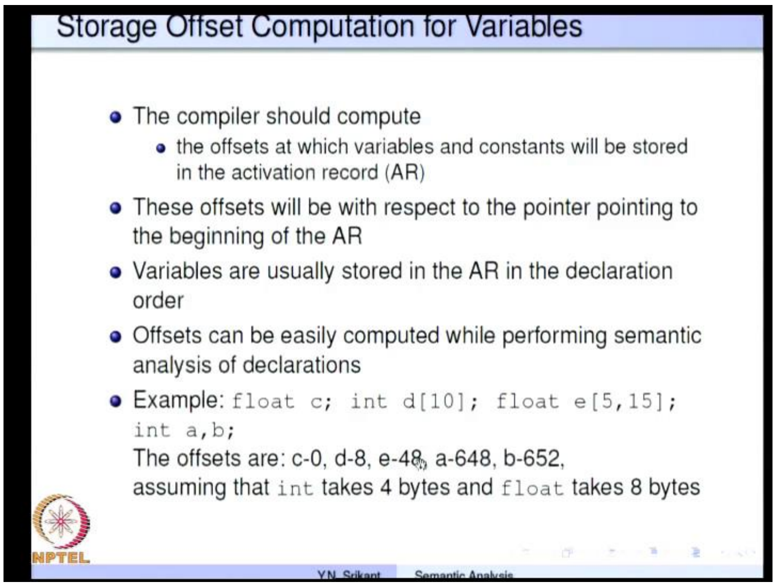
In the productions here we had actions in between also, but for the productions here we have an action only at the end of this particular production. Ideally we could have actually broken this production into two, so i d array goes to some you know non terminal say a, followed by another non terminal b, the non terminal a goes to i d and then you introduce the search and if found in that case. So, this is possible, but you know it would probably issue an error message which closed to i d, but otherwise since the dimension list is usually quite small it is not a list of expressions, but it is only a list of constants. Producing an error message after the entire array declaration is over is still not too bad, then the productions becomes straightforward it need not be broken into two and that is the approach we have taken in this production.

So, let us see how the dimension list produces various ranges of the array. So, one possibility is DIMLIST goes to num; that means, the production terminates the list of dimensions. The second one is DIMLIST going to num comma DIMLIST; that means, we are producing a list of dimension 1, 2, 3, etcetera. For the last one we use the production DIMLIST going to num.

So, in this case DIMLIST dot pointer is a list made out of num dot value, so make list makes a list out of the argument. So, num dot value is the range of that particular dimension not the first, but that particular dimension. So, DIMLIST goes to num comma DIMLIST, so we have produced you know a dimension and rest of the dimension will be produced from DIMLIST 2 here. So, there is already a DIMLIST pointer available for DIMLIST 2, we have a new dimension which is produced here, so DIMLIST 1 dot ptr will be set as a combination of num dot value, that is this particular value and DIMLIST 2 dot pointer. So, this value is appended to this particular dimension.

So, this way we produce, but the order is important we must retain this first and then this. So, the concatenation should not be such that num dot value appears at the end of the dimension list 1 dot pointer, it must actually appear right in the beginning. So, the order is very important, because these are the various ranges of the array. So, this was about producing the rather analyzing the variables and there decelerations, the error messages given works quite simple, the most of the error messages would actually be in the usage part.

(Refer Slide Time: 12:15)



Storage Offset Computation for Variables

- The compiler should compute
  - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations
- Example: `float c; int d[10]; float e[5,15]; int a,b;`
  The offsets are: c-0, d-8, e-48, a-648, b-652,
  assuming that `int` takes 4 bytes and `float` takes 8 bytes

Y.N. Srikant Semantic Analysis

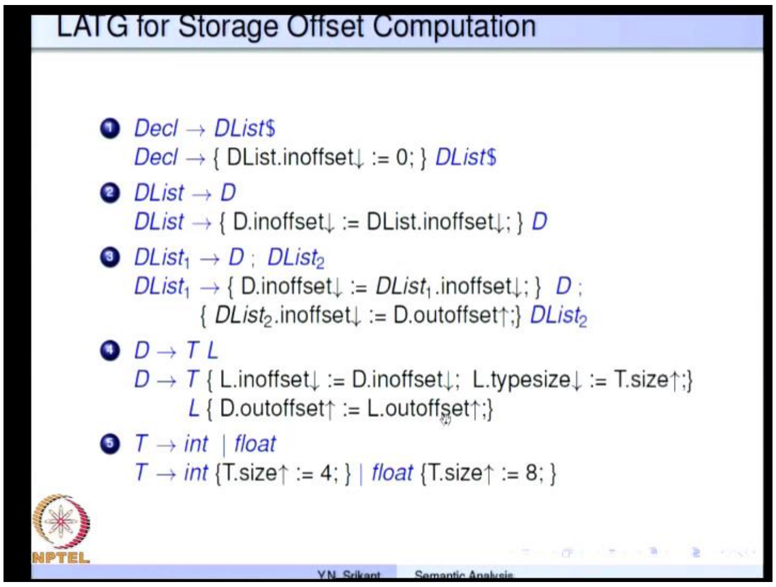
But before doing that we have to look at a very important computation called as the storage computation for variables, which is actually related to the decelerations and the list of decelerations to be very specific, so let us see what these are. The compiler is supposed to compute the offsets at which variables and constants will be stored in the activation record on the stack. So, as you perhaps know when the programs starts running activation record where for various procedures are created on the stack and each activation record holds the data of that particular function or procedure. So, there is a stack pointer and there is a pointer pointing to the most recent activation record.

So, we need to compute the offsets at which variables and constants will be stored in these activation records for that particular procedure. The offsets will always you know starts with 0, but then depending on the specific format of the activation record the code generator has to add a particular extra value in order to produce correct address. These offsets will be with respect to the pointer pointing to the beginning of the activation record, so that is what I said, they will all be starting from 0.

Variables are usually stored in the activation record in the deceleration order. So, if we have a deceleration float c, into d 10, float e 5 comma 15 and then int a comma b. The order in which the variables will be allocated space on the activation record would be first c and then d, then e, then a, then b, so this is how they would be assigned locations in the activation record. Offsets can be easily computed while performing semantic analysis of declarations, so the basic idea is you know described here. So, assuming that this is our deceleration, the offsets for the first variable is always 0, so the offset for c is 0. Assuming that int takes 4 bytes and float takes 8 bytes, so we add 8 to the offset of c, so that we get 8 and that would be the offset for d, so d gets an offset of 8.

The 10 elements of d, which occupy 10 into 4, 40 bytes would stretch from 8 onwards, the offset 8 onwards. So, because d takes 40 bytes and we have an initial you know offset of 8 for d, the offset for e will obviously be 40 plus 8, that is 48. Now, e has you know 75 elements, 15 into 5, each of which takes 8 bytes, so that would be 600 and we add 600 you know to this 48, so we get 648 as the offset for e, for a sorry. So, because e has an offset of 48 and e itself has a size of 600 bytes, so a gets 648 as its offset, add 4 to it and you get the offset for b that is 652. So, this is precisely the competition that we need to do, whenever we have a list of decelerations we keep accumulating the offset and then add it to the next deceleration in order to get the new offset.

(Refer Slide Time: 16:03)



LATG for Storage Offset Computation

1. $Decl \rightarrow DList\$$
   $Decl \rightarrow$ { DList.inoffset↓ := 0; } $DList\$$
2. $DList \rightarrow D$
   $DList \rightarrow$ { D.inoffset↓ := DList.inoffset↓; } D
3. $DList_1 \rightarrow D\ ;\ DList_2$
   $DList_1 \rightarrow$ { D.inoffset↓ := $DList_1$.inoffset↓; } D ;
   { $DList_2$.inoffset↓ := D.outoffset↑;} $DList_2$
4. $D \rightarrow T\ L$
   $D \rightarrow T$ { L.inoffset↓ := D.inoffset↓; L.typesize↓ := T.size↑;}
   L { D.outoffset↑ := L.outoffset↑;}
5. $T \rightarrow int\ |\ float$
   $T \rightarrow int$ {T.size↑ := 4; } | $float$ {T.size↑ := 8; }

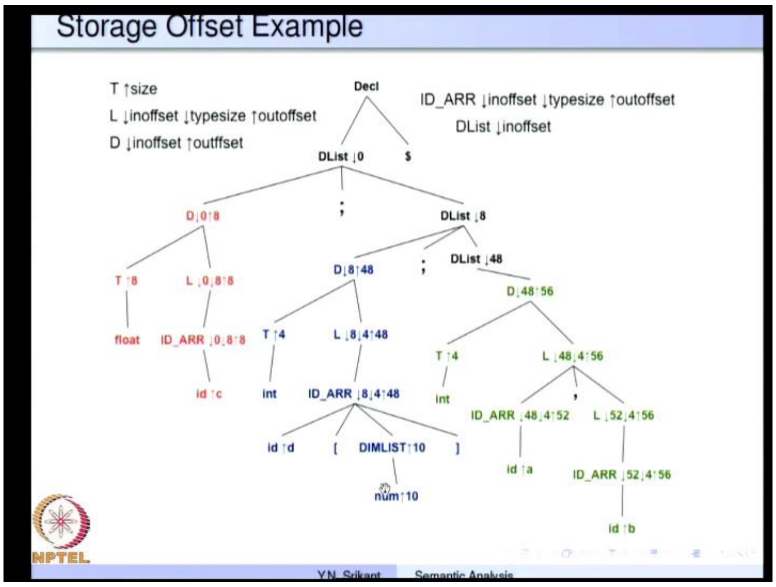
Y.N. Srikant   Semantic Analysis

Let us see how it is done for our deceleration grammar. We have a an inherited attribute called D List dot inoffset, which is initialize to 0 at the top, because as I said all offsets actually are start with a value 0 and then we increment them. So, in this production D List going to D, we transfer the offset D List dot inoffset to D dot inoffset, because this is just one deceleration and then call the procedure for D.

In the production D List going D semicolon D List, the offset from here is transferred to D, so D dot inoffset is equal to D List 1 dot inoffset. Then the deceleration D has certain variables with various sizes, so the total size is added to the incoming offset and that would be outgoing offset of D and that is transferred to D List 2. So, after D we have D List 2 dot inoffset equal to D dot outoffset, so D produces the new offset which can be assigned to D List 2 and then D List 2 is called.

So, this is a standard strategy, we always produce you know we take a inoffset, add the size of variables that our deceleration produces and then transfer that as the outoffset. So, D going to T L, T produces its size, so T dot size and L gets inoffset through D, so L dot inoffset equal to D dot inoffset, then the typesize of L is T dot size. So, the list of identifiers produced by L you know will have this L will have an inherited attribute L dot typesize which is nothing but T dot size. Now, L produces identifiers, so it adds variable list size to the inoffset and that would become D dot outoffset. So, T dot size is set to either 4 or 8 depending on the variable type being integer or float.

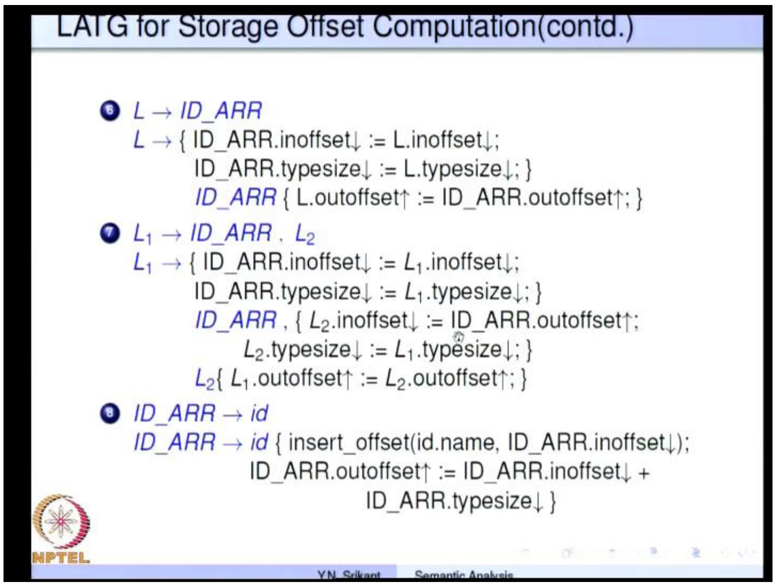
(Refer Slide Time: 18:41)


Storage Offset Example

So, let us took at an example in part, so we start with declaration which goes to D List and dollar, this goes to D semicolon D List, so we have a 0 initialization here, which is passed on to this D and then the T here gets its size from float and the 0 is passed on to L. So, you can see that L has inoffset, typesize and outoffset in that order. So, 0 is you know given to L as inoffset and that is passed on to the next levels, that we are going to see very soon. So, let us consider this part of the example later. So, again here the incoming offset is 8, which is given by D List and how did it get its incoming offset as a 8? It actually got it from D, you know D produced 8 as its outoffset and that was pushed into D List as its inoffset that you can see here.

So, D List dot inoffset is D dot outoffset and that is sent to this D, which again intern goes about sending into this L and L send it into I D array, this D List produces a D which again goes to T and L, so this L gets it 48 and then it takes it further. So, this is how the inherited attributes start from the top and flow to the non terminals below.

(Refer Slide Time: 20:28)



LATG for Storage Offset Computation(contd.)

- $L \rightarrow ID\_ARR$
  $L \rightarrow$ { ID_ARR.inoffset↓ := L.inoffset↓;
  ID_ARR.typesize↓ := L.typesize↓; }
  ID_ARR { L.outoffset↑ := ID_ARR.outoffset↑; }
- $L_1 \rightarrow ID\_ARR\ ,\ L_2$
  $L_1 \rightarrow$ { ID_ARR.inoffset↓ := L_1.inoffset↓;
  ID_ARR.typesize↓ := L_1.typesize↓; }
  ID_ARR , { L_2.inoffset↓ := ID_ARR.outoffset↑;
  L_2.typesize↓ := L_1.typesize↓; }
  L_2{ L_1.outoffset↑ := L_2.outoffset↑; }
- $ID\_ARR \rightarrow id$
  $ID\_ARR \rightarrow id$ { insert_offset(id.name, ID_ARR.inoffset↓);
  ID_ARR.outoffset↑ := ID_ARR.inoffset↓ +
  ID_ARR.typesize↓ }

Y.N. Srikant Semantic Analysis

So, let us continue our example. So, L going I D array and L going to I D array comma L 2 these are the to which produce the list of declarations with a certain type t. So, the L dot type size you know will already be available. So, I D array dot type size is initialize to L dot type size I D array dot in offset is initialize to L dot inoffset.

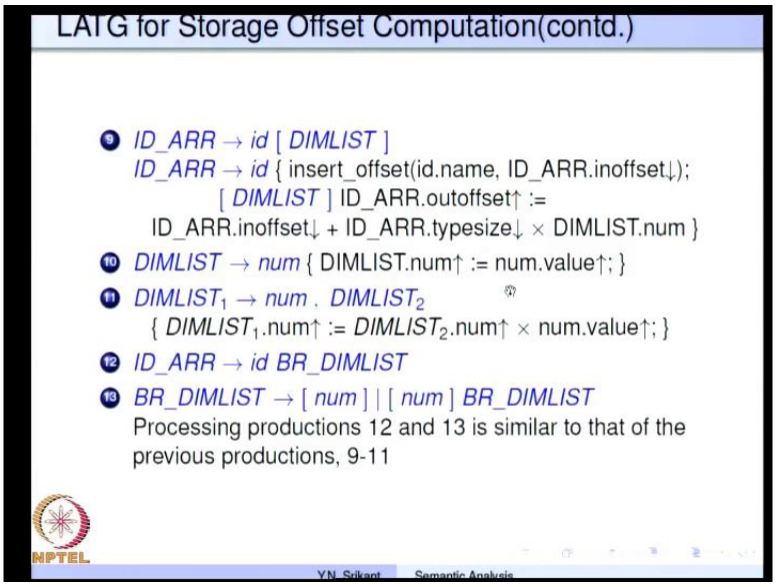
And then I D array is called finally, L dot offset outoffset is the synthesis attribute of L that is made as I D array dot outoffset. So, whatever I D array produces is transfer to L what about this case again the inherited attributes of L 1 some of them are transfer to I D array and L 2 for example, typesize I D array dot typesize is also L 1 dot typesize and L 2 dot typesize is also L 1 dot typesize, whereas the inherited attribute inoffset is copied from L 1. So, I D array dot inoffset is L 1 dot inoffset and then I D array is called. So, the outoffset of I D R 2 goes in as inoffset of L 2, that we saw already here.

So, L 2 dot inoffset equal to I D array dot outoffset and then L 2 is called and L 1 dot outoffset is made as L 2 dot outoffset. I D array itself produces either an I D or you know a array. So, when it produces a simple variable I D what we really do is insert the offset into the symbol table for the corresponding name I D dot name and the offset being I D array dot inoffset. So, you we must remember here that the semantic analysis part which deals with error messages etcetera appear before this. So, part is executed only if there are no other error messages given out.

I D array dot outoffset you know the, this name as a certain size. And what is that size? I D array dot inoffset plus I D array dot sorry I D array dot typesize is the size of this particular name. So, what is the out offset it is the sum of inoffset plus typesize that is assigned to outoffset. So, let us see how the example proceeds here as well. So, from L the 0 is transferred to I D array and then the name is transferred from I D 2 I D array it is introduce into the symbol table, and since this I D is of type the float the size 8 of the float actually gets inherited into L comes down to I D array. And then it combines with the inoffset and produces an outoffset of 8, which is passed on to L which is intern passed on to d. And that intern goes into dlist.

So, dlist passes it on to this D which is actually 8 here right. So, this is the size of the int which is 4 that is inherited into L. So, 8 and 4 are both inherited into L which are passed on to I D array. So, this we are going to see in the next slide. So, here dlist you know passes the inoffset 48 into D which is again passed to L this 4 is also passed into L. So, this L produces I D array comma l. So, I D array gets inoffset of 48 and typesize of 4 with the combination of these 2 it inserts the name I a into symbol table, and its outoffset is 52 48 plus 4 and that would be you know the inherited attribute of this L. So, 52 comes here. And then this I D array gets 52 it also gets 4 which is the nothing but the typesize here and at this point this name gets the inoffset 52 its size is 4. So, the outoffset is 52 which is synthesis and produce sent to L which intern traverses all the way up to this point.

(Refer Slide Time: 25:13)



LATG for Storage Offset Computation(contd.)

9. $ID\_ARR \rightarrow id$ [$DIMLIST$]
   $ID\_ARR \rightarrow id$ { insert_offset(id.name, ID_ARR.inoffset$\downarrow$);
   [$DIMLIST$] ID_ARR.outoffset$\uparrow$:=
   ID_ARR.inoffset$\downarrow$ + ID_ARR.typesize$\downarrow$ $\times$ DIMLIST.num }
10. $DIMLIST \rightarrow num$ { DIMLIST.num$\uparrow$:= num.value$\uparrow$; }
11. $DIMLIST_1 \rightarrow num$, $DIMLIST_2$
    { $DIMLIST_1$.num$\uparrow$:= $DIMLIST_2$.num$\uparrow$ $\times$ num.value$\uparrow$; }
12. $ID\_ARR \rightarrow id\ BR\_DIMLIST$
13. $BR\_DIMLIST \rightarrow$ [num] | [num] $BR\_DIMLIST$
    Processing productions 12 and 13 is similar to that of the previous productions, 9-11

Y.N. Srikant Semantic Analysis

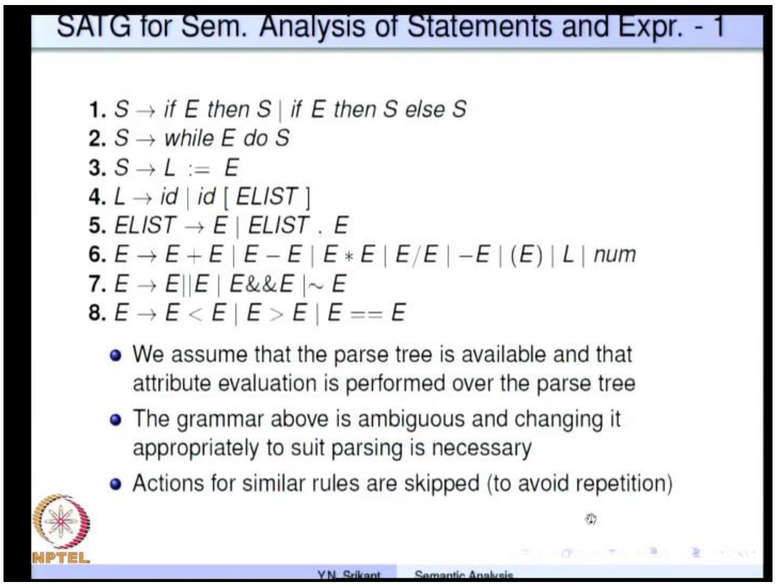
So, let us see how the arrays are processed. So, I D array produces I D followed by a dimension list, the I D inoffset inserted into the symbol table rather offset inoffset inserted into the symbol table with I D array dot inoffset as the incoming offset. When the dimension list inoffset passed and now we produce I D array dot outoffset as I D array dot you know rather.

So, whatever coming in into I D array inoffset the inoffset then we consider the I D array dot typesize and finally, multiplied by it dimension list dot num. So, what inoffset dimension list dot num that inoffset actually the number of elements in the array. So, that when multiplied by the element size of the array, which in bytes gives you the total size of the array and that is added to the incoming offset and that would produce a the outgoing offset I D array dot outoffset.

Dimension list either is a single number or a sequence of numbers separated by a comma. So, if n is a symbol number we just have dimension list dot num equal to num dot value, when it is a sequence of numbers dimension list 1 dot num that is a synthesis attribute is dimension list 2 dot num multiplied by num dot value because this is the earlier dimension. So, if we have a 10 comma 15. So, each element of the dimension is of the size 15. So, 10 into 15 is the size of the array and that is what we are doing here this is the rest of the array this is the four most dimension of the array as for as this list is concern. So, we take the dimension list 2 dot num multiplied by the num value that is gives us the total size of the array. So, for as this particular list is concerned.

This processing the same array deceleration with a different format is very similar to these to nothing very special. So, we are going to skip that part. So, we looked up the examples up to this point. So, the incoming offset is 8 the type of the array is 4 bytes in size. So, here we have a single dimensional array. So, dimlist 10 produces sorry num 10 produces a dimlist 10. So, number of elements is 10 here and that multiplied by 4 gives us 40 added to 8 the incoming offset gives us 48 that is the outgoing offset of I D array which traverses all way up to this point and goes down as I already discussed.

(Refer Slide Time: 28:09)



SATG for Sem. Analysis of Statements and Expr. - 1

1. $S \rightarrow if\ E\ then\ S \mid if\ E\ then\ S\ else\ S$
2. $S \rightarrow while\ E\ do\ S$
3. $S \rightarrow L\ :=\ E$
4. $L \rightarrow id \mid id\ [\ ELIST\ ]$
5. $ELIST \rightarrow E \mid ELIST\ ,\ E$
6. $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \mid L \mid num$
7. $E \rightarrow E || E \mid E \&\& E \mid \sim E$
8. $E \rightarrow E < E \mid E > E \mid E == E$

- We assume that the parse tree is available and that attribute evaluation is performed over the parse tree
- The grammar above is ambiguous and changing it appropriately to suit parsing is necessary
- Actions for similar rules are skipped (to avoid repetition)

So, now we come to the semantics analysis of statements and expressions. So, let us see how this is done. So, we have seen decelerations so far. Now, let us look at the body of the code a couple of assumptions here, we are assuming that there are no scopes for the names. In other words there are no blocks within the statements. So, decelerations you know processing decelerations with within blocks etcetera is a little more complicated, and we will do it a little later. For the present we are assuming that there is only one block and that block has decelerations earlier, which we have already seen. And then there are statements of this kind for example, there is a if then statement and also a if then else statement there is a while do statement, there is an assignment statement.

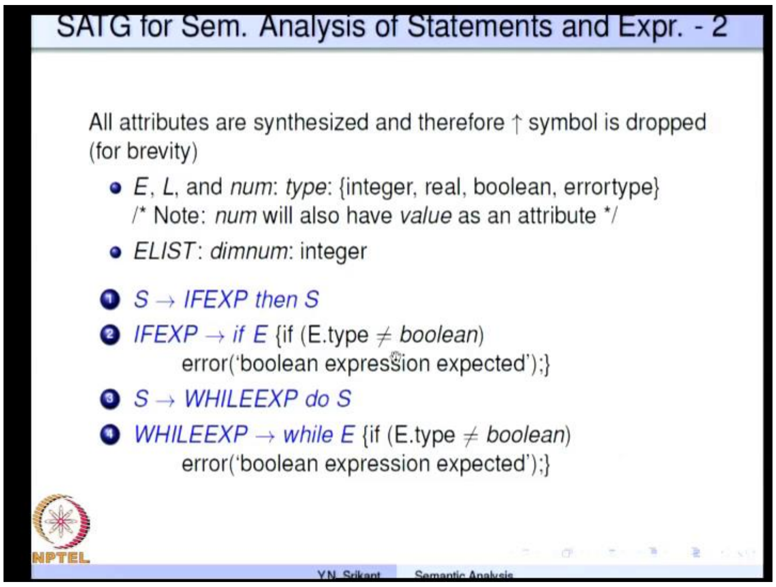
And the left hand side of the assignment can be just a simple I D or an array element. The dimensions of the array are produced the using elist going to E or elist comma E, expressions are of various types. So, E plus E E minus E E star E E slash E minus E paraenesis expression, then L and num. So, these are the various possibilities for expressions. So, remember there is no E going to I d, but L produces I D that is a very important observation necessary, otherwise we will assume that there no I D that is wrong actually. And E can also be of you know Boolean expression type. So, these are all arithmetic expressions and these are all Boolean expressions E or E E and E and not e.

And we also have expressions which compare other expressions. So, E less than E E greater than E or E E equal to E. There are many other possibilities for relational operators, but processing them will be very similar. So, let us consider the very basic relational operators and then it does not matter, what others are the processing will be similar. A very important observation here is that we cannot separate the arithmetic expressions from either the Boolean expressions or the relational expressions. So, this is necessary for writing a proper grammar, the context free grammars cannot differentiate between arithmetic expressions Boolean expressions and relational expressions.

So, it will really permits all types of identifiers in each of these places for example, it will permit array minus array which is not permitted in C. And here it may actually put characters or arrays again and here also it can put you know any type operands including arrays which are not permitted. So, the semantic analysis routine must look at the 2 operands, look at the operator and then decide whether theory you know the arithmetic operation should be allowed at all or an error message should be given. The same as to be done for the Boolean operators and the relational operators. Again this grammar is ambiguous so we will assume that modifications to a make it L a L R 1 are possible and that would be left for exercises.

The parse tree is assumed to be available and will assume that the attribute evaluation is parser performed over the parser tree. We will also skip rules for similar productions for example, we will handle E going to E plus E and skip these that is because they are all you know very similar. Similarly we will handle E going E or E and skip the other 2, we will handle E less than E and skip the other 2.

(Refer Slide Time: 32:37)


SATG for Sem. Analysis of Statements and Expr. - 2

All attributes are synthesized and therefore ↑ symbol is dropped (for brevity)

- *E*, *L*, and *num*: *type*: {integer, real, boolean, errortype}
  /* Note: *num* will also have *value* as an attribute */
- *ELIST*: *dimnum*: integer

1. *S* → *IFEXP then S*
2. *IFEXP* → *if E* {if (E.type ≠ *boolean*)
   error('boolean expression expected');}
3. *S* → *WHILEEXP do S*
4. *WHILEEXP* → *while E* {if (E.type ≠ *boolean*)
   error('boolean expression expected');}

Y.N. Srikant Semantic Analysis

So, this is an SATG you know. So, we already saw an example of LATG for decelerations. So, let us see how synthesis attributes are used in type checking for statements. So, all the attributes are synthesized and therefore, we will drop the arrow symbol because it is just extra every attribute will have this.

So, we are going to skip it what are the various attributes expression and L and num have an attribute called type the type can be either integer or real or Boolean or error type. So, error type is new. So, this is indicative of error Boolean type is possible only during you know the parsing of expressions. So, it cannot be a specified by the programmer integer and real are the only ones which are specified by the programmer. Num will also; obviously, have value as an attribute, which is of no use to us right now, but it will be useful later. Elist will have dimnum as an attribute, which is of type the integer. So, a number of you know the dimensions and things like that.

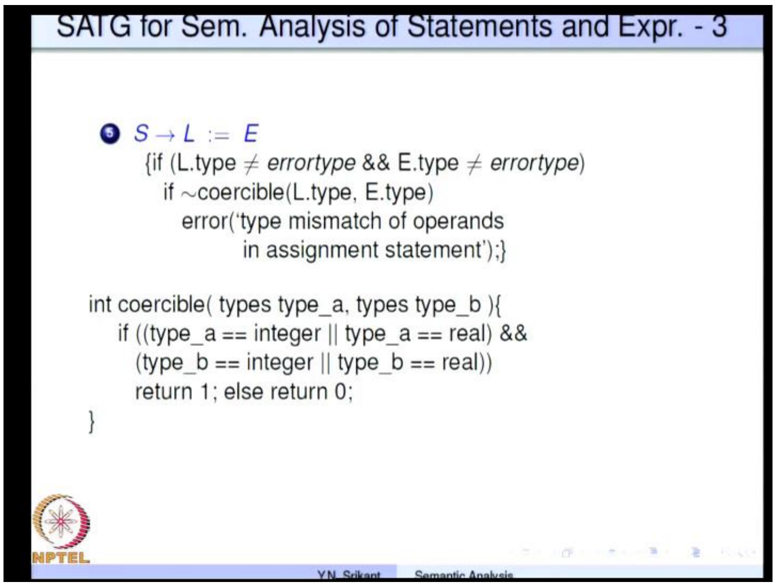
So, let us consider the expressions 1 by 1, the sorry the productions 1 by 1 s going to ifexp then S. This particular production actually would appear as if E then S otherwise if E then else S and so on, right? We have chosen to break the production into 2 the first part is ifexp then S the second part is ifexp going to if E, why did we really do it I hinted at something like this for you know array, you know generations of variables which are arrays, decelerations which are arrays. The reason is suppose we had a single production

D going to if E then S and we attach a semantic action right here saying if E dot type is not Boolean then error Boolean expression expected.

It is possible that the expression itself is very large and it is also possible that S is a fairly large body of statements. So, if we attach the you know this semantic check and the error is actually given out it would be actually given probably half a page later because the entire this entire if E then s would be passed and only then this semantic action would have been executed. So, whereas, here the as soon as if and E are passed this semantic action gets executed. So, the as soon as it finds that the expression parsed is a not a Boolean expression the error message is printed out at the right place instead of providing the error message after S is also completely parsed.

So, this is all you know that is necessary while doing semantic analysis you must make sure that the error message is actually given where it is relevant. The situation is similar here also we could have said S going to while E do S, but then if s is a very large body of statements this error would have been issued after half a page or 1 page of code. So, we break the production into 2 parts s going to whileexp and do S and other one being whileexp going to while E. And the semantic check is the same if E dot type is not Boolean then Boolean expression expected error message is issued. If there is no error message then there semantic analysis continuous. So, it is possible to do this type of attribute competition during L R parsing provided the grammar is not you know ambiguous.

(Refer Slide Time: 36:56)



SATG for Sem. Analysis of Statements and Expr. - 3

```
● S → L := E
    {if (L.type ≠ errortype && E.type ≠ errortype)
      if ~coercible(L.type, E.type)
        error('type mismatch of operands
              in assignment statement');}

int coercible( types type_a, types type_b ){
    if ((type_a == integer || type_a == real) &&
      (type_b == integer || type_b == real))
      return 1; else return 0;
}
```

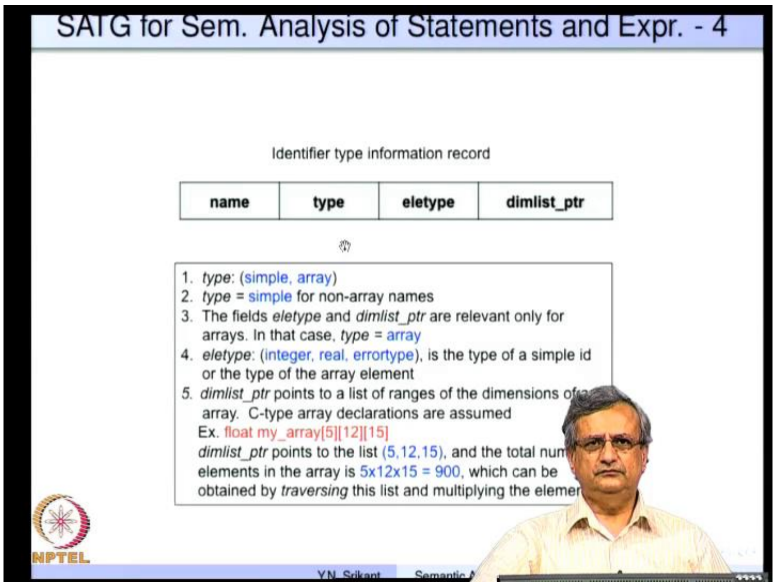
Y.N. Srikant Semantic Analysis

Let us take a simple assignment L equal to E. So, now L would be a large expression E would be similarly fairly large expression, there could have been a semantic error in the midst of any of these expressions. So, it is possible that L dot has become errortype and it is also possible that E dot type has become errortype, one or both have been become errortype in the process of analyzing both L and E. So, if they are errortypes then you know the there is nothing to do really because L dot type is errortype and E dot type is errortypes. So, there is nothing to do no more error messages need to be given out.

But suppose both are perfectly you know honorable types and they are not of type errortype, then we check whether it is possible to convert L dot type to E dot type or vice versa, why should we do this? The point is L is the left hand side of an assignment E is the right hand side of the assignment. So, if E is of type say integer and L is of type float then we must convert int into float and then generate code for the assignment. So, the coercible function checks whether such coercions in type or permitted are possible etcetera. So, if it cannot be converted then there is an error message given out, type mismatch of operands in assignment statement.

So, what does the function coercible do? It takes 2 operands type a and type b checks is type a integer or is it real. So, if it is one of these then it is a valid type, is type b integer or is it a real if it is one of these then that is also a valid type otherwise they would be error type. So, if it is 1 if these 2 are satisfied together that is why the and return by success by returning a 1, otherwise return is 0. So, not coercible implies 0. So, coercible implies 1. So, in the in case the coercible function returns a 0 we print out a an error message.

(Refer Slide Time: 39:48)



SATG for Sem. Analysis of Statements and Expr. - 4

Identifier type information record

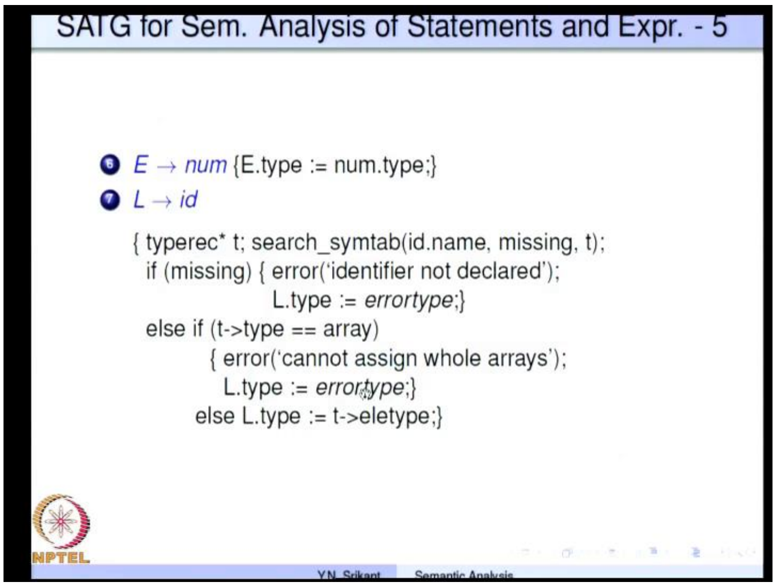
name	type	eletype	dimlist_ptr

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, errortype), 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 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 num elements in the array is 5x12x15 = 900, which can be obtained by *traversing* this list and multiplying the elemen

Y.N. Srikant Semantic A

Again I am showing the type information record just to show you that the number of elements in the array, you know rather the number of dimensions of the array will have to be considered, when we do semantic analysis that is what we are doing to do next.

(Refer Slide Time: 40:10)



SATG for Sem. Analysis of Statements and Expr. - 5

```
E → num {E.type := num.type;}
L → id
 { typerec* t; search_symtab(id.name, missing, t);
   if (missing) { error('identifier not declared');
                  L.type := errortype;}
   else if (t->type == array)
           { error('cannot assign whole arrays');
             L.type := errortype;}
        else L.type := t->eletype;}
```

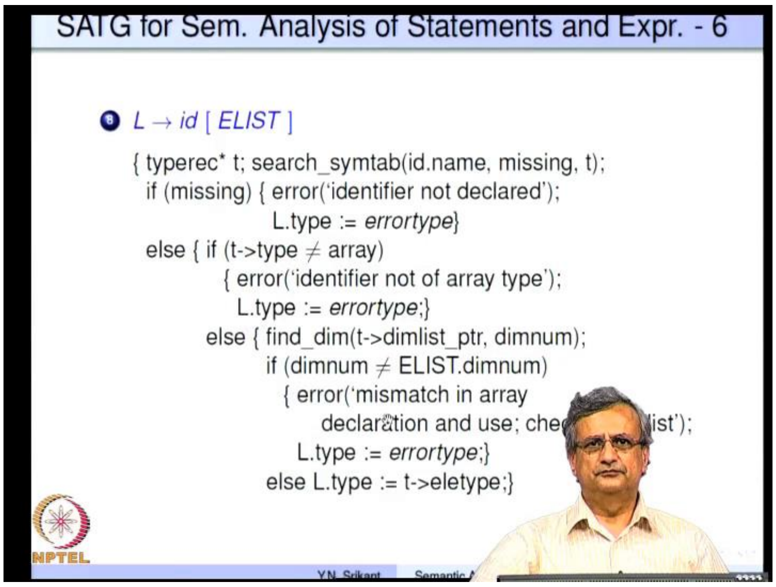
Y.N. Srikant Semantic Analysis

E going to num so E dot type is num dot type there is much to do here, whatever is the type of this number either integer or by mistake a float or a character etcetera the type of that particular number is transferred as E dot type. What about L going to I d? So, if there is an identifier it is always necessary to search the symbol table find out whether the name exist in the symbol table. If its exist then we need to extract the time information of that particular identifier.

So, for that purpose we have type extract T this is the same temporary type information record search a symbol table with the name of the identifier, the flag missing and the temporary t. So, if the flag missing is true then the name is not present in the symbol table and if the flag missing is falls, then the name indeed present in the symbol table. So, if missing is true we issue an error message identifier not declared and assign the type as L dot type equal to errortype. So, this is how L would get the type information during its parsing and semantic analysis phase, else if T pointer type is array. So, we have really got the symbol table you know rather than a name, and its information from the symbol table.

So, we know the type of that particular identifier here this name does not have any array subscripts following it. So, it must be a simple name. So, if T dot type is array give an error message cannot assign whole arrays. So, we do not permit that in this particular small grammar. So, it is possible to permit assignment of 1 array to another array or parts of arrays to another array and so on, but our language does not permit it at this point if we want to permit such extra behavior, the semantic analysis has to be modified appropriately. So, if the identifier is of type error we assign errortype because there is an error it has to be of simple type, otherwise L dot type is T pointer eletype. So, eletype would be either integer or real and that is what is really obtained here. So, T dot type is already L dot type T dot type is already simple. So, L dot type would be T pointer eletype.

(Refer Slide Time: 43:18)



SATG for Sem. Analysis of Statements and Expr. - 6

- $L \rightarrow id$ [*ELIST*]

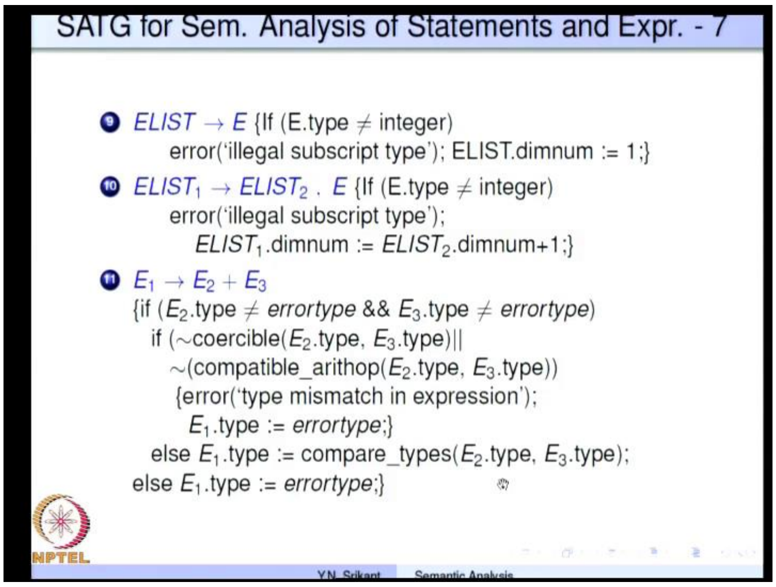
```
{ typerec* t; search_symtab(id.name, missing, t);
  if (missing) { error('identifier not declared');
                 L.type := errortype}
  else { if (t->type ≠ array)
            { error('identifier not of array type');
              L.type := errortype;}
         else { find_dim(t->dimlist_ptr, dimnum);
                if (dimnum ≠ ELIST.dimnum)
                   { error('mismatch in array
                       declaration and use; che         list');
                     L.type := errortype;}
                else L.type := t->eletype;}
```

Now, we come to the semantic analysis of arrays. So, we have L going to identifier followed by an expression list. We assume that you know we do not assign arrays to arrays, but we can only assign to an element of the array or assign an element of the array to some other variable. So, this Elist covers all the dimensions of the name I D, typerec star T as usual we decelerate at a typerec you know a record of type typerec and a pointer to it. Search the symbol table for I D dot name missing an T as before, if missing issue an error message and make L dot type errortype and there ends the matter.

Otherwise in this case this I D is supposed to be of type array so is T pointer type not equal to array if it is not then there is a error. So, identifier not of array type, in the previous case we issue an error message if it was an error because this was supposed to be a ordinary identifier. So, if it indeed an error we find the dimensions of the array, the parameters of this find dim function are T pointer dimlist pointer. So, we are pointing to the dimlist pointer of the name and the number of dimensions is provided in dimnum.

So, if dim num that is obtained from the symbol table is not equal to the number of expressions present in Elist dot dim num that is here, whatever subscripts are available here we count them and make them available as Elist dot dim num. If these two do not match it is possible that we have return less number of subscripts here or it is also possible that we have return more number of subscripts here than necessary. So, in both cases there is a miss match in array deceleration induce check index list, that is a message which is printed out. L dot type in that case becomes error type. So, if everything is then we assign L dot type as T pointer eletype again eletype would be integer or real.

(Refer Slide Time: 45:54)



SATG for Sem. Analysis of Statements and Expr. - 7

```
9  ELIST → E {If (E.type ≠ integer)
        error('illegal subscript type'); ELIST.dimnum := 1;}
10 ELIST_1 → ELIST_2 , E {If (E.type ≠ integer)
        error('illegal subscript type');
          ELIST_1.dimnum := ELIST_2.dimnum+1;}
11 E_1 → E_2 + E_3
   {if (E_2.type ≠ errortype && E_3.type ≠ errortype)
     if (~coercible(E_2.type, E_3.type)||
        ~(compatible_arithop(E_2.type, E_3.type))
        {error('type mismatch in expression');
          E_1.type := errortype;}
     else E_1.type := compare_types(E_2.type, E_3.type);
   else E_1.type := errortype;}
```

Y.N. Srikant Semantic Analysis

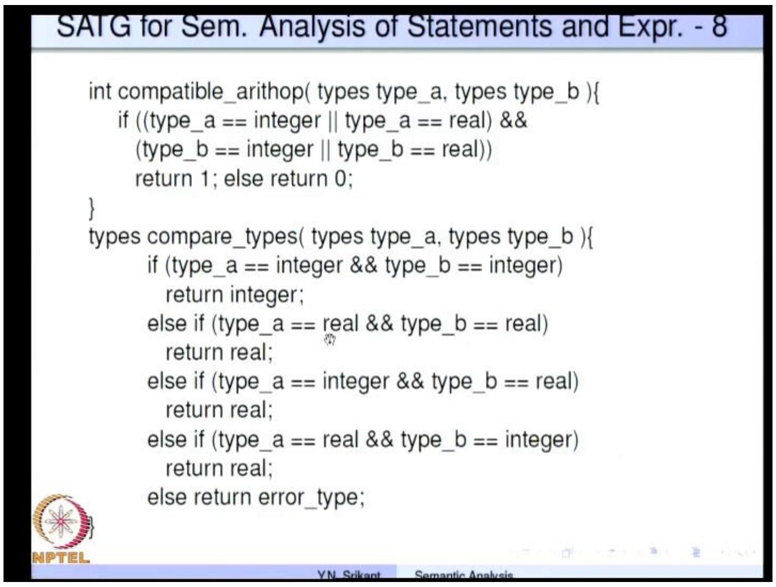
So, how do we count the dimensions elist going to E. So, or Elist going to Elist comma E. So, in the case of Elist going to E if E dot type is not integer subscripts have to evaluate to integer type they cannot becomes floats then the error message illegal subscript type is provided, and a list dot dimnum becomes 1. The reason is whenever we when we use this would actually be the first dimension first subscript of the array subscription list its because Elist 2. Let us there are there is a three dimensional array we would use Elist going to Elist comma E once then this Elist would be again expanded as Elist comma E and the third Elist would be actually expanded to E.

So, whenever we this particular production we would be looking at the first dimension of the array. So, that is why it is to say dimnum equal to 1 at this point here this would have actually seen a certain number of dimensions and this is adding another dimension. Here so Elist 1 dot dimnum equal to Elist 2 dot dimnum plus 1 is the correct estimate of the number of dimensions in. So, far and; obviously, E dot type R equal to integer essential otherwise an error messages given out.

What we should observe further here is even if 1 of the dimensions creates an error, the semantic analyzer does not stop it actually goes to the next dimension, and continues the semantic analysis. This is a very desirable feature because the parser or the compiler is supposed to give out as many errors as possible error messages as possible in 1 parser of the entire program. So, if we catch just 1 error and then asks the program on to repair it and run it you know run through it run it through the complier all over again to get the next error, possibly nobody would use our complier.

So, now, we go to a very important part of this semantic analysis you know semantic analysis of expressions, this plus is indicative as I said it can be replace by minus star slash etcetera. So, in the case of expressions of this type, these are arithmetic expressions and in the following slides we will look at Boolean and relational expressions. So, if E 2 dot is not equal to error type and E 3 dot type not equal to errortype, only then we do this part of it otherwise we simple set E 1 dot type as errortype and get out because what is the point in doing semantic analysis further, if either 1 or both these is errors erroneous type. So, assuming that both are valid types and not error types, next we check whether E 2 dot type and E 3 dot type you know are compatible with each other they can be converted from 1 to another. So, let us see what that means.

(Refer Slide Time: 49:41)



SATG for Sem. Analysis of Statements and Expr. - 8

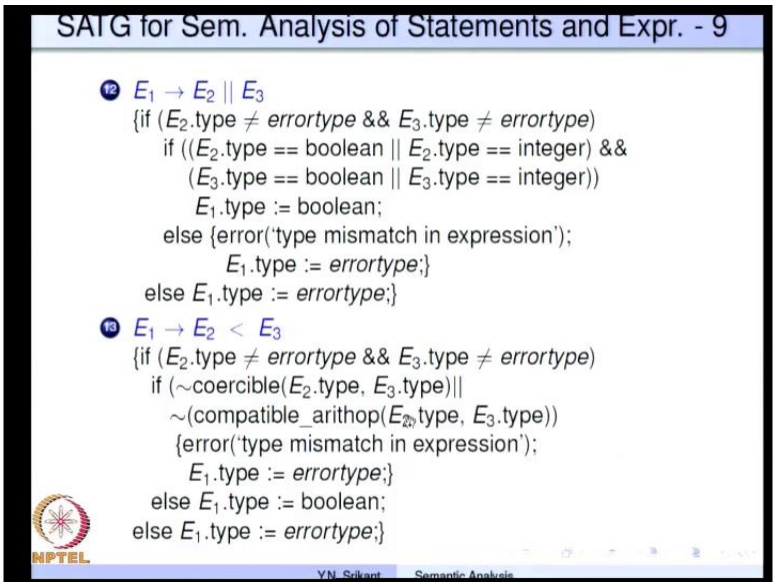
```
int compatible_arithop( types type_a, types type_b ){
    if ((type_a == integer || type_a == real) &&
      (type_b == integer || type_b == real))
      return 1; else return 0;
}
types compare_types( types type_a, types type_b ){
        if (type_a == integer && type_b == integer)
          return integer;
        else if (type_a == real && type_b == real)
          return real;
        else if (type_a == integer && type_b == real)
          return real;
        else if (type_a == real && type_b == integer)
          return real;
        else return error_type;
}
```

Y.N. Srikant Semantic Analysis

So, the sorry coercible we already saw compatible arithop is what we need to see later, coercible simply says if E 2 dot types here, yes coercible simply says if first one is either integer or real and a second one is a either integer or real then return 1 else 0. So, in this case it is possible to keep the operands as both real one of them real and the other one as you know this can be int and this can be real or this can be real and this can be int both can be int both can be real, these are the 4 possibilities.

So, among these int plus int is easy to handle real plus real is also easy to handle, if 1 is int and the other is real we can still handle it nothing wrong with it. Similarly if this is int and this is real we can still handle this expression and there is nothing wrong with it. So, that is what this coercible checks. If the coercible part of it is falls we also of course, need to check whether the arithmetic operator and the 2 operands are compatible with each other. Let us see what this compatible arithop is? Compatible arithop says take 2 types if type a is integer or type a is real similarly type b is integer or type b is real return 1 else 0.

(Refer Slide Time: 51:24)



SATG for Sem. Analysis of Statements and Expr. - 9

```
12 $E_1 \rightarrow E_2 \mid\mid E_3$
   {if ($E_2$.type $\neq$ errortype && $E_3$.type $\neq$ errortype)
       if (($E_2$.type == boolean || $E_2$.type == integer) &&
          ($E_3$.type == boolean || $E_3$.type == integer))
           $E_1$.type := boolean;
       else {error('type mismatch in expression');
               $E_1$.type := errortype;}
     else $E_1$.type := errortype;}
13 $E_1 \rightarrow E_2 < E_3$
   {if ($E_2$.type $\neq$ errortype && $E_3$.type $\neq$ errortype)
      if (~coercible($E_2$.type, $E_3$.type)||
        ~(compatible_arithop($E_2$.type, $E_3$.type))
         {error('type mismatch in expression');
          $E_1$.type := errortype;}
      else $E_1$.type := boolean;
    else $E_1$.type := errortype;}
```

Y.N. Srikant Semantic Analysis

So, we actually permit other possibilities here you know, but our explanation at this point of time does not contain them. So, we may say look in a see for example, this could become a string you know so this could become a string rather a single character, this could also become an integer. So, both these can still be added because characters can be converted 2 integer. So, all these possibilities exist in the real language c, but in our language we assume that the only four possibilities permitted or int int int real real int and real real.

So, these are all compatible with the plus operation. So, compatible arithop checks for these. In fact, these 2 are very similar the only reason we have said separately is the bigger complier can actually expand the you know coercible and compatible arithop functions to add many possibilities. So, in general a keeping them separate shows the structure of the complier little more clearly.

So, if it is neither coercible nor compatible arithop then a type mismatch in a expression has happened an error message is issued. So, E 1 dot type in that case becomes error type. So, otherwise if everything is E 1 dot type becomes compare type E 2 dot type comma E 3 dot type. So, what does that really do? So, this takes the 2 types you know if they are depending on whether they are integer the other one being type b being integer then it returns integer if 1 of them both of them are real then returns real, if 1 is integer the other 1 is real it returns real. Similarly first one is real and second one is integer then also it returns real.

The reason is simple if we adding a floating point number and an integer the integer its need to be converted to floating point because range of floating point numbers is much larger than that of integers. So, that is what is done here next is the Boolean operation and finally, the relational operator, this is fairly straightforward if neither or errortype then we go ahead otherwise we set E 1 dot type as errortype. E 2 dot type can be Boolean E three dot type can be integer, or the other possibility E 3 dot type is Boolean E 3 dot type is integer. So, if both these old then E 1 dot type would be Boolean.

So, in other words we do not mind having int or Boolean Boolean or int int or int Boolean or Boolean all these four possibilities are permitted by this combination. So, if none of these are true then type mismatch in a expression is given out, and E 1 dot type is set as error type if these conditions hold then E 1 dot is set as Boolean. And instead of or we could have used and also in this case the last 1 is E E 1 going to E 2 less than E 3. So, in this case if there not errortypes we check whether they are.

So, in this case this less than is very similar to an arithmetic operator because we can only you know compare either Booleans or integers usually sorry or the arithmetic a numbers. So, these have to be numbers. So, we cannot have any other type here. So, that is why we check coercible and then compatible arithop, if neither of them is true then we generate an error message otherwise E 1 dot type is termed as Boolean, this is the only difference between E 2 plus E 3 and E 2 less than E 3. Finally, E 1 dot type really becomes Boolean. So, let us stop here and continue analysis of see in the next lecture.

Thank you.