# Semantic Analysis with Attribute Grammars
# Part 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# 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

# 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. $BR\_DIMLIST \rightarrow [ \; num \; ] \mid [ \; num \; ] \; BR\_DIMLIST$

# 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

# 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.

# 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. $L \rightarrow \{ID\_ARR.type \downarrow := L.type \downarrow\}$ *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);}
}
```

# LATG for Sem. Analysis of Variable Declarations - 4

4. *ID_ARR* $\rightarrow$ *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)}
   }
5. *DIMLIST* $\rightarrow$ *num*
   {DIMLIST.ptr↑ := makelist(num.value↑)}
6. $DIMLIST_1 \rightarrow num,\ DIMLIST_2$
   {$DIMLIST_1.ptr\uparrow$ := append(num.value↑, $DIMLIST_2.ptr\uparrow$)}

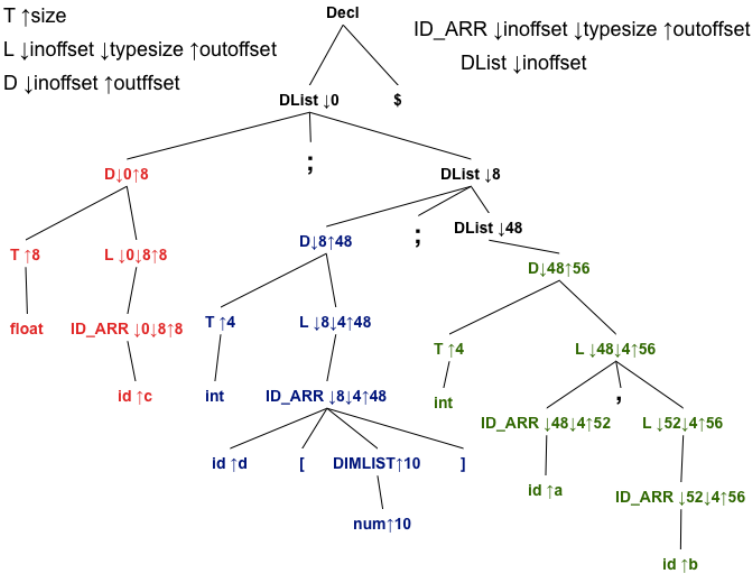
# Storage Offset Computation for Variables

- 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

# 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. $T \rightarrow int \mid float$
   $T \rightarrow int$ {T.size↑ := 4; } | $float$ {T.size↑ := 8; }

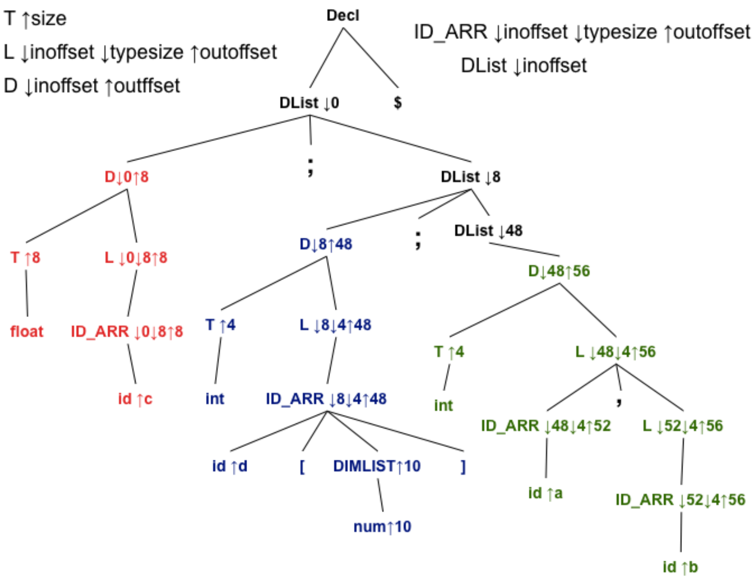
# Storage Offset Example



# LATG for Storage Offset Computation(contd.)

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

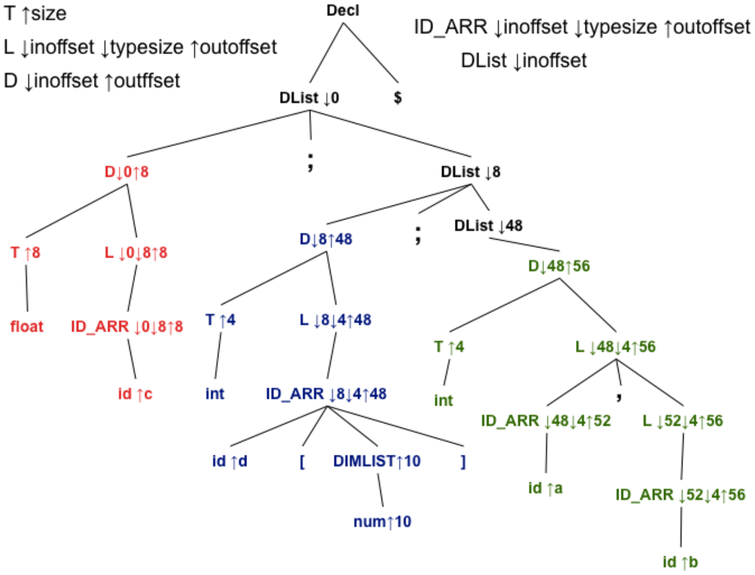
# Storage Offset Example



# LATG for Storage Offset Computation(contd.)

9. $ID\_ARR \rightarrow id$ [ $DIMLIST$ ]
   $ID\_ARR \rightarrow id$ { insert_offset(id.name, ID_ARR.inoffset↓);
   [ $DIMLIST$ ] ID_ARR.outoffset↑ :=
   ID_ARR.inoffset↓ + ID_ARR.typesize↓ × DIMLIST.num }
10. $DIMLIST \rightarrow num$ { DIMLIST.num↑ := num.value↑; }
11. $DIMLIST_1 \rightarrow num$ , $DIMLIST_2$
    { $DIMLIST_1$.num↑ := $DIMLIST_2$.num↑ × num.value↑; }
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

# Storage Offset Example



# 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)

# 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 $\neq$ *boolean*)
   error('boolean expression expected');}
3. *S* → *WHILEEXP do S*
4. *WHILEEXP* → *while E* {if (E.type $\neq$ *boolean*)
   error('boolean expression expected');}

5. $S \rightarrow L := E$

   {if (L.type $\neq$ *errortype* && E.type $\neq$ *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;
}
```

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.

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

6. $E \rightarrow num$ {E.type := num.type;}
7. $L \rightarrow id$

```
{ 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;}
```

8. $L \rightarrow id \; [ \; ELIST \; ]$

```
{ 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; check index list');
                    L.type := errortype;}
                else L.type := t->eletype;}
```

9. $ELIST \rightarrow E$ {If (E.type $\neq$ integer)
   error('illegal subscript type'); ELIST.dimnum := 1;}
10. $ELIST_1 \rightarrow ELIST_2$ , $E$ {If (E.type $\neq$ integer)
   error('illegal subscript type');
   $ELIST_1$.dimnum := $ELIST_2$.dimnum+1;}
11. $E_1 \rightarrow E_2 + E_3$
   {if ($E_2$.type $\neq$ *errortype* && $E_3$.type $\neq$ *errortype*)
   if ($\sim$coercible($E_2$.type, $E_3$.type)||
   $\sim$(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*;}

# 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;
}
```

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 ($\sim$coercible($E_2$.type, $E_3$.type)||
      $\sim$(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*;}