Semantic Analysis - 5

LATG for Sem. Analysis of Variable Declarations

$$Decl \rightarrow DList\$$$

$$DList \rightarrow D \mid D \; ; \; DList$$

$$D \rightarrow T \; L$$

$$T \rightarrow int \mid float$$

$$L \rightarrow ID\_ARR \mid ID\_ARR \; , \; L$$

$$ID\_ARR \rightarrow id \mid id \; [\; DIMLIST \;] \mid id \; BR\_DIMLIST$$

$$DIMLIST \rightarrow num \mid num, \; DIMLIST$$

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

Note: array declarations have two possibilities

```
int a[10,20,30]; float b[25][35];
```

Identifier Type Information in 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

$L_1 \rightarrow$ {ID_ARR.type$\downarrow$:= L_1.type$\downarrow$ } ID_ARR ,
{L_2.type$\downarrow$:= L_1.type$\downarrow$;} L_2

$L \rightarrow \{ID\_ARR.type \downarrow := L.type \downarrow\}$ ID_ARR

$ID\_ARR \rightarrow id$

```
{ 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

$ID\_ARR \rightarrow id$ [$DIMLIST$]

{ search ...; if (found) ...;

else { typerec* t; t->type := array;

t->eletype := ID_ARR.type$\downarrow$;

t->dimlist_ptr := DIMLIST.ptr$\uparrow$;

insert_symtab(id.name$\uparrow$, t)}

}

$DIMLIST \rightarrow num$

{DIMLIST.ptr$\uparrow$:= makelist(num.value$\uparrow$)}

$DIMLIST_1 \rightarrow num,\ DIMLIST_2$

$\{DIMLIST_1.ptr \uparrow$:= append(num.value$\uparrow$, $DIMLIST_2.ptr \uparrow$)}

LATG for Storage Offset Computation

- 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

$Decl \rightarrow DList\$$

$Decl \rightarrow \{\ \mathrm{DList.inoffset}{\downarrow} := 0;\ \}\ DList\$$

$DList \rightarrow D$

$DList \rightarrow \{\ \mathrm{D.inoffset}{\downarrow} := \mathrm{DList.inoffset}{\downarrow};\ \}\ D$

$DList_1 \rightarrow D\ ;\ DList_2$

$DList_1 \rightarrow \{\ \mathrm{D.inoffset}{\downarrow} := DList_1.\mathrm{inoffset}{\downarrow};\ \}\ \ D\ ;$
$\{\ DList_2.\mathrm{inoffset}{\downarrow} := \mathrm{D.outoffset}{\uparrow};\}\ DList_2$

$D \rightarrow T\ L$

$D \rightarrow T\ \{\ \mathrm{L.inoffset}{\downarrow} := \mathrm{D.inoffset}{\downarrow};\ \ \mathrm{L.typesize}{\downarrow} := \mathrm{T.size}{\uparrow};\}$
$L\ \{\ \mathrm{D.outoffset}{\uparrow} := \mathrm{L.outoffset}{\uparrow};\}$

$T \rightarrow int\ \mid\ float$

$T \rightarrow int\ \{\mathrm{T.size}{\uparrow} := 4;\ \}\ \mid\ float\ \{\mathrm{T.size}{\uparrow} := 8;\ \}$

LATG for Storage Offset Computation

$L \rightarrow ID\_ARR$

$L \rightarrow$ { ID_ARR.inoffset↓ := L.inoffset↓;
ID_ARR.typesize↓ := L.typesize↓; }
ID_ARR { L.outoffset↑ := ID_ARR.outoffset↑; }

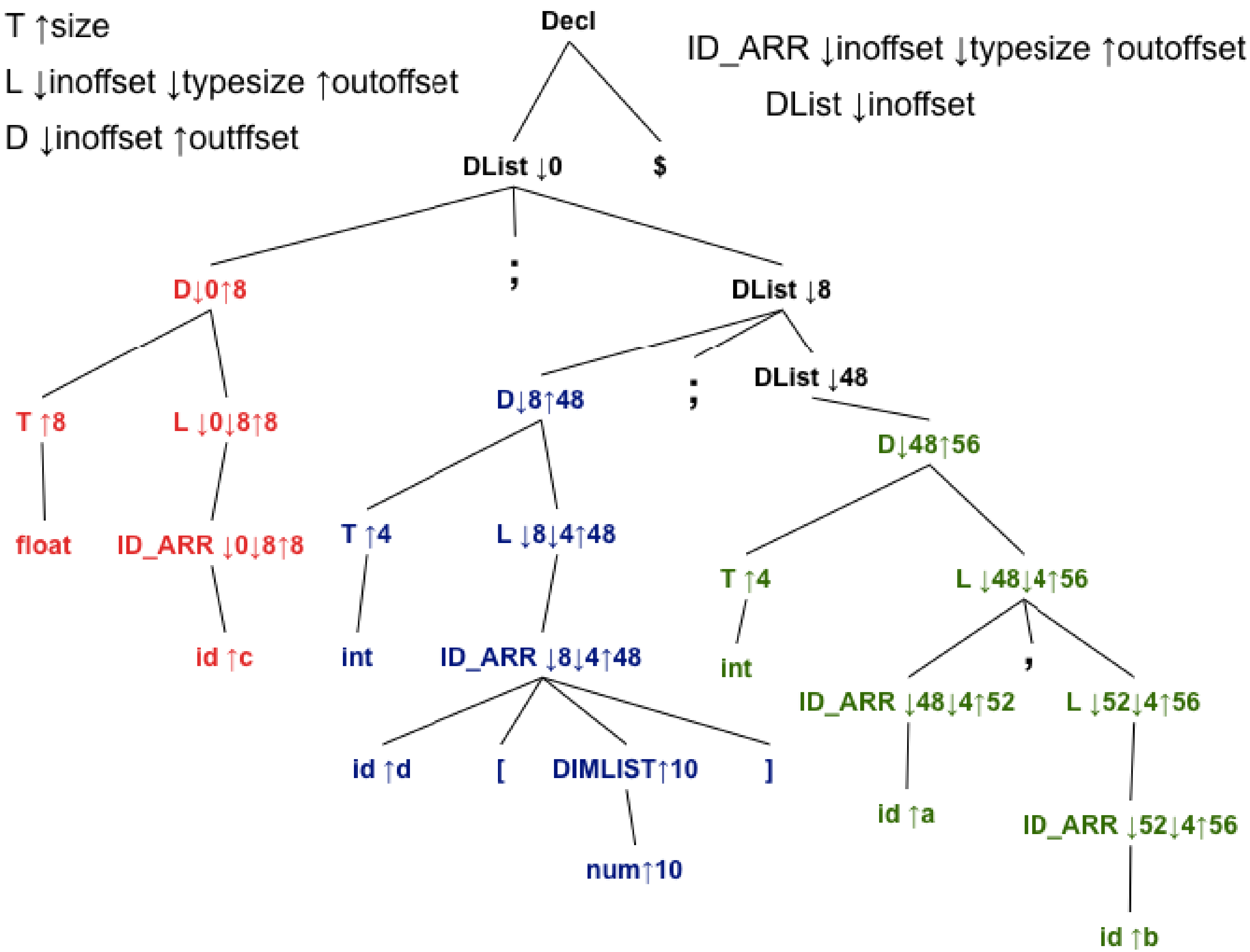
$L_1 \rightarrow ID\_ARR\ ,\ L_2$

$L_1 \rightarrow$ { 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↓ }

LATG for Storage Offset Computation - Example



LATG for Storage Offset Computation

$ID\_ARR \rightarrow id$ [$DIMLIST$]

$ID\_ARR \rightarrow id$ { insert_offset(id.name, ID_ARR.inoffset$\downarrow$);
[$DIMLIST$] ID_ARR.outoffset$\uparrow$:=
ID_ARR.inoffset$\downarrow$ + ID_ARR.typesize$\downarrow$ $\times$ DIMLIST.num }

$DIMLIST \rightarrow num$ { DIMLIST.num$\uparrow$:= num.value$\uparrow$; }

$DIMLIST_1 \rightarrow num\ ,\ DIMLIST_2$
{ $DIMLIST_1$.num$\uparrow$:= $DIMLIST_2$.num$\uparrow$ $\times$ num.value$\uparrow$; }

$ID\_ARR \rightarrow id\ BR\_DIMLIST$

$BR\_DIMLIST \rightarrow$ [num] | [num] $BR\_DIMLIST$

Processing productions 12 and 13 is similar to that of the previous productions, 9-11

LATG for Storage Offset Computation - Example

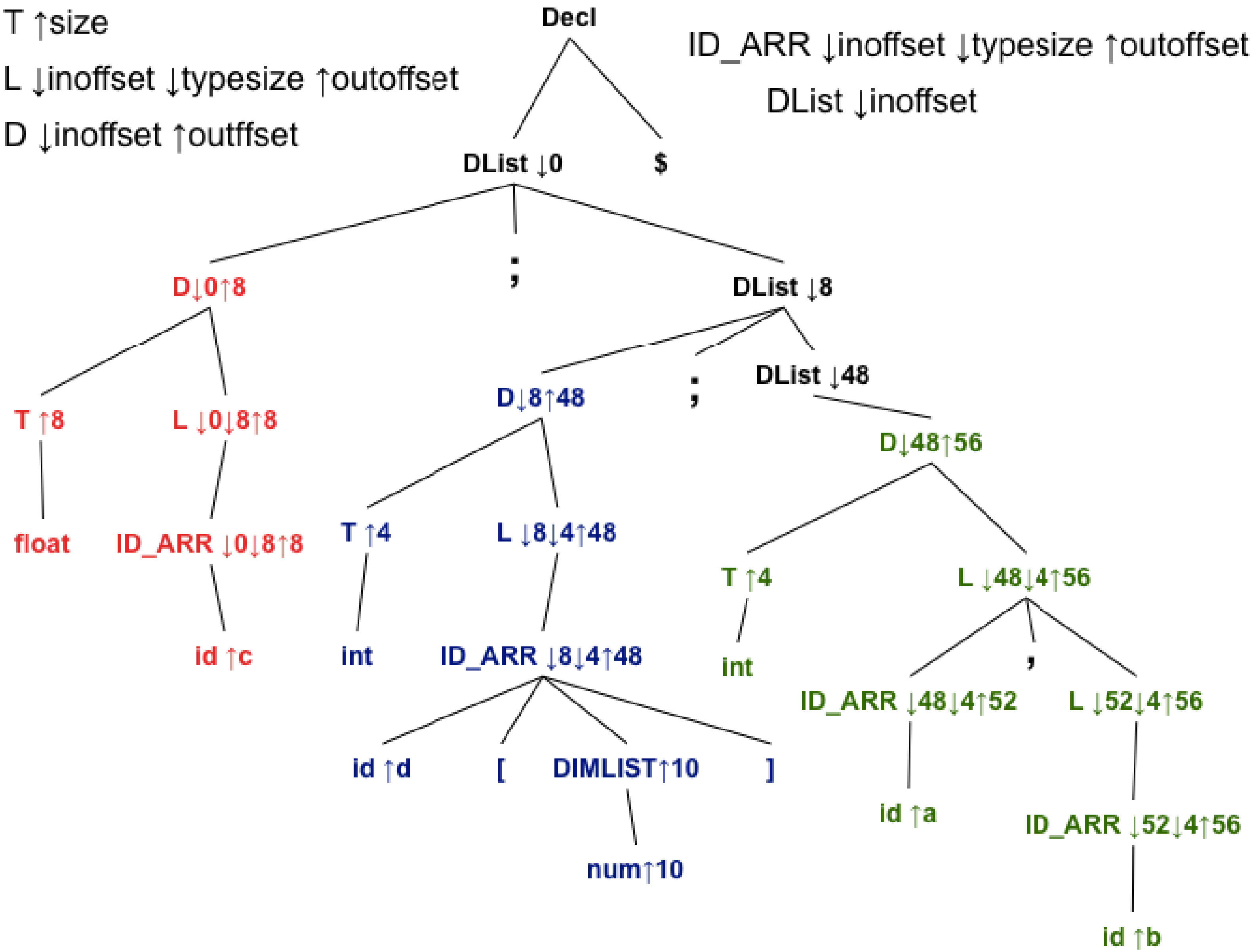
T ↑size

L ↓inoffset ↓typesize ↑outoffset

D ↓inoffset ↑outffset

ID_ARR ↓inoffset ↓typesize ↑outoffset

DList ↓inoffset



SATG for Sem. Analysis of Statements and Expressions

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 Expressions

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

ELIST: *dimnum*: integer

S → *IFEXP then S*

IFEXP → *if E* {if (E.type ≠ *boolean*)
error('boolean expression expected');}

S → *WHILEEXP do S*

WHILEEXP → *while E* {if (E.type ≠ *boolean*)
error('boolean expression expected');}

SATG for Sem. Analysis of Statements and Expressions

$S \rightarrow L := E$

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

SATG for Sem. Analysis of Statements and Expressions

$E \rightarrow num$ {E.type := num.type;}

$L \rightarrow id$

```
{ 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;}
```

SATG for Sem. Analysis of Statements and Expressions

$L \rightarrow id$ [$ELIST$]

```
{ 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;}
```

SATG for Sem. Analysis of Statements and Expressions

$ELIST \rightarrow E$ {If (E.type $\neq$ integer)
error('illegal subscript type'); ELIST.dimnum := 1;}

$ELIST_1 \rightarrow ELIST_2\ ,\ E$ {If (E.type $\neq$ integer)
error('illegal subscript type');
$ELIST_1$.dimnum := $ELIST_2$.dimnum+1;}

$E_1 \rightarrow E_2 + E_3$
{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 Expressions

```
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;
}
```

SATG for Sem. Analysis of Statements and Expressions

$E_1 \rightarrow E_2 \mid\mid E_3$

{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*;}

$E_1 \rightarrow E_2 < E_3$

{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*;}