Semantic Analysis - 4

LAG Example 2

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

LAG Example 2

$$S \longrightarrow E \ \{E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow\}$$

$$E_1 \longrightarrow E_2 + T \ \{E_2.symtab \downarrow := E_1.symtab \downarrow;$$
$$E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow\}$$

$$E \longrightarrow T \ \{T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow\}$$

$$E_1 \longrightarrow let\ id = E_2\ in\ (E_3)$$
$$\{E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow;$$
$$E_3.symtab \downarrow := E_1.symtab \downarrow \setminus \{id.name \uparrow \rightarrow E_2.val \uparrow\}\}$$

$$T_1 \longrightarrow T_2 * F \ \{T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$$
$$T_2.symtab \downarrow := T1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow\}$$

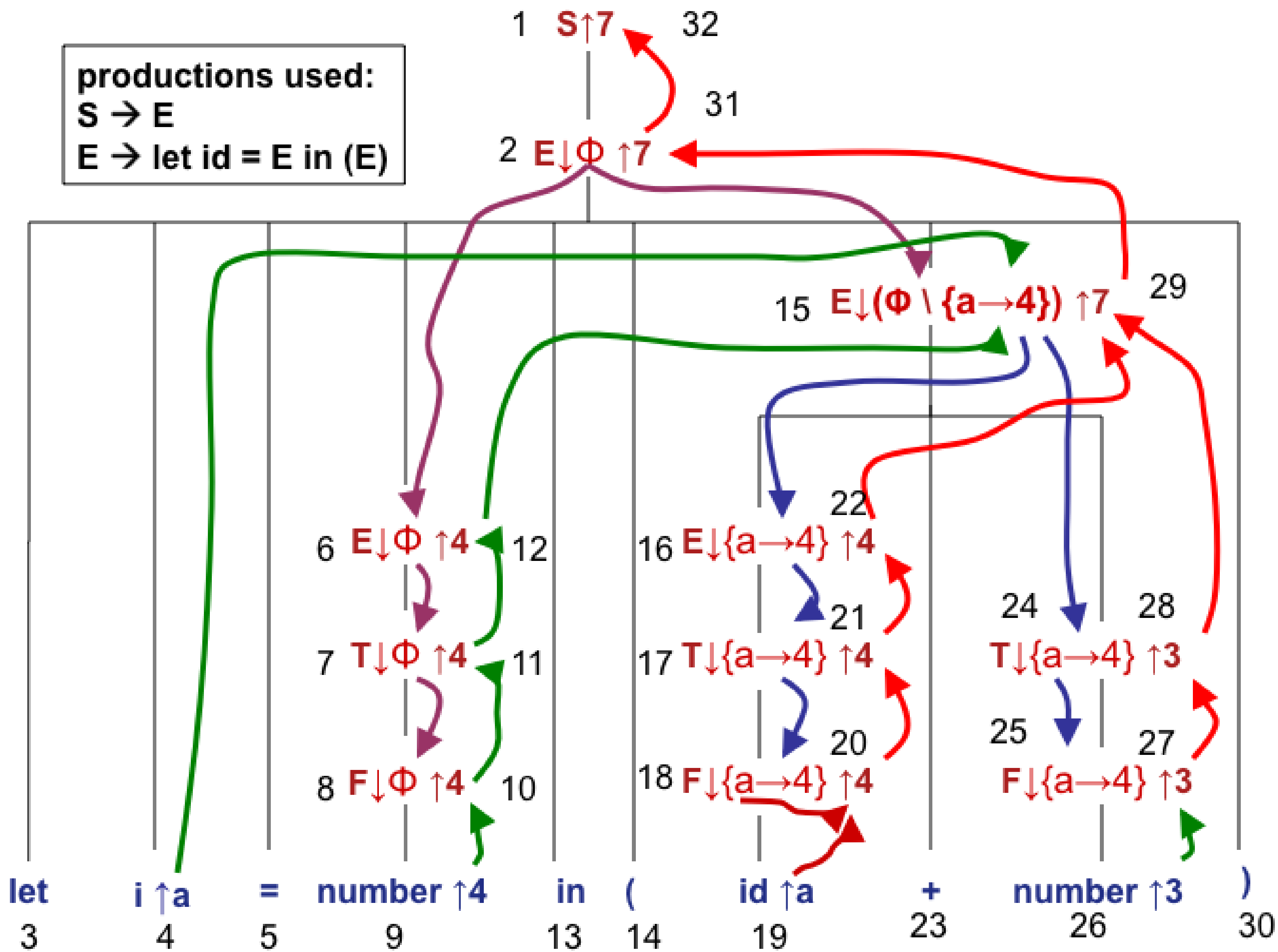
$$T \longrightarrow F \ \{T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow\}$$

$$F \longrightarrow (E) \ \{F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow\}$$

$$F \longrightarrow number \ \{F.val \uparrow := number.val \uparrow\}$$

$$F \longrightarrow id \ \{F.val \uparrow := F.symtab \downarrow [id.name \uparrow]\}$$

LAG Example 2 – Attribute Evaluation Order



Attributed Translation Grammar

- Apart from attribute computation rules, some program segment that performs either output or some other side effect-free computation is added to the AG

- Eg.: symbol table operations, writing generated code to a file, etc.

- To incorporate these actions, evaluation orders may be constrained

- Constraints are added to 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

SATG for a Desk Calculator

```
%%
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 */
     ;
%%
```

SATG for a Modified Desk Calculator

```
%%
lines: lines expr '\n' {printf("%g\n",$2);}
     | lines '\n'
     | /* empty */
     ;
expr : NAME '=' expr {sp = symlook($1);
       sp->value = $3; $$ = $3;}
     | NAME {sp = symlook($1);$$ = sp->value;}
     | expr '+' expr {$$ = $1 + $3;}
     | expr '-' expr {$$ = $1 - $3;}
     | expr '*' expr {$$ = $1 + $3;}
     | expr '/' expr {$$ = $1 - $3;}
     | '(' expr ')' {$$ = $2;}
     | NUMBER /* type double */
     ;
%%
```

LAG and LATG for Declarations

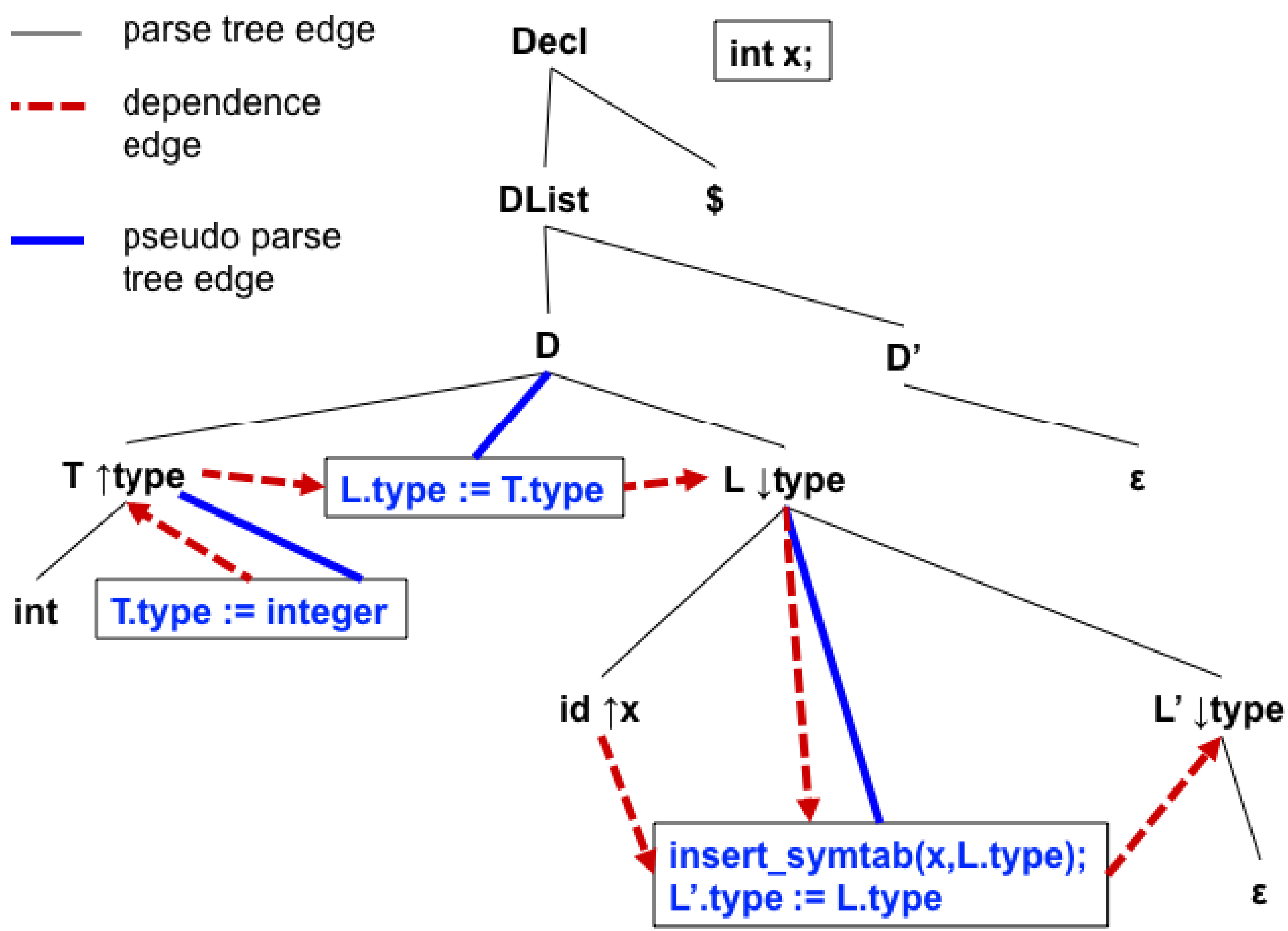
LAG (notice the changed grammar)

1. $Decl \rightarrow DList\$$ **2.** $DList \rightarrow D\ D'$ **3.** $D' \rightarrow \epsilon\ |; DList$
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. $L' \rightarrow \epsilon\ |\ , L\ \{L.type \downarrow := L'.type \downarrow;\}$
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.** $D' \rightarrow \epsilon\ |; DList$
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. $L' \rightarrow \epsilon\ |\ , \{L.type \downarrow := L'.type \downarrow;\}\ L$

LATG for Declarations – Dependence Graph



Integrating LATG into RD Parser

```
/* 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();
}
```

Integrating LATG into RD Parser

```
/* L --> id {insert_symtab(id.name, L.type);
              L'.type := L.type} L' */
void L(vartype type){if mytoken.token == ID
            {insert_symtab(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 ; }
```

SATG for Declarations

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. $L_1 \rightarrow L_2\ ,\ id\ \{sp = insert\_symtab(id.name \uparrow);$
   $L_1.namelist \uparrow = append(L_2.namelist \uparrow, sp);\}$

SATG for 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) }
```

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↓;
         t->dimlist_ptr := DIMLIST.ptr↑;
         insert_symtab(id.name↑, t)}
}
```

$DIMLIST \rightarrow num$

{DIMLIST.ptr↑ := makelist(num.value↑)}

$DIMLIST_1 \rightarrow num,\ DIMLIST_2$

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