Semantic Analysis with Attribute Grammars Part 3

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
- Attributed translation grammars
- Semantic analysis with attributed translation 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)

L-Attributed and S-Attributed Grammars

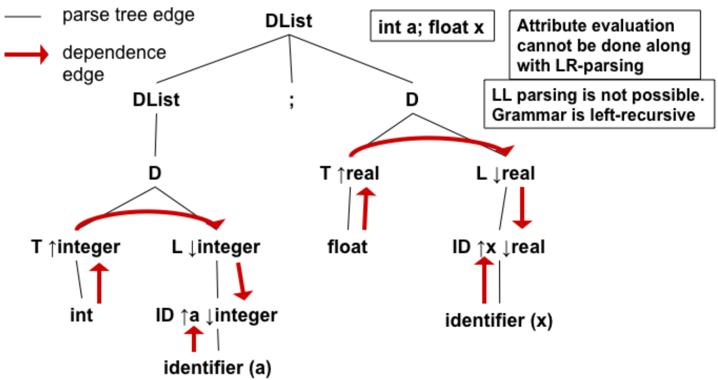
- An AG with only synthesized attributes is an S-attributed grammar
  - Attributes of SAGs can be evaluated in any bottom-up order over a parse tree (single pass)
  - Attribute evaluation can be combined with LR-parsing (YACC)
- In L-attributed grammars, attribute dependencies always go from *left to right*
- More precisely, each attribute must be
  - Synthesized, or
  - Inherited, but with the following limitations:
    consider a production $p: \ A \rightarrow X_1 X_2 ... X_n$. Let $X_i.a \in AI(X_i)$.
    $X_i.a$ may use only
    - elements of $AI(A)$
    - elements of $AI(X_k)$ or $AS(X_k)$, $\ k = 1, ..., i-1$
      (*i.e.,* attibutes of $X_1, ..., X_{i-1}$)
- We concentrate on SAGs, and 1-pass LAGs, in which attribute evaluation can be combined with LR, LL or RD parsing

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
}
```

Example of LAG - 1



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

Example of Non-LAG

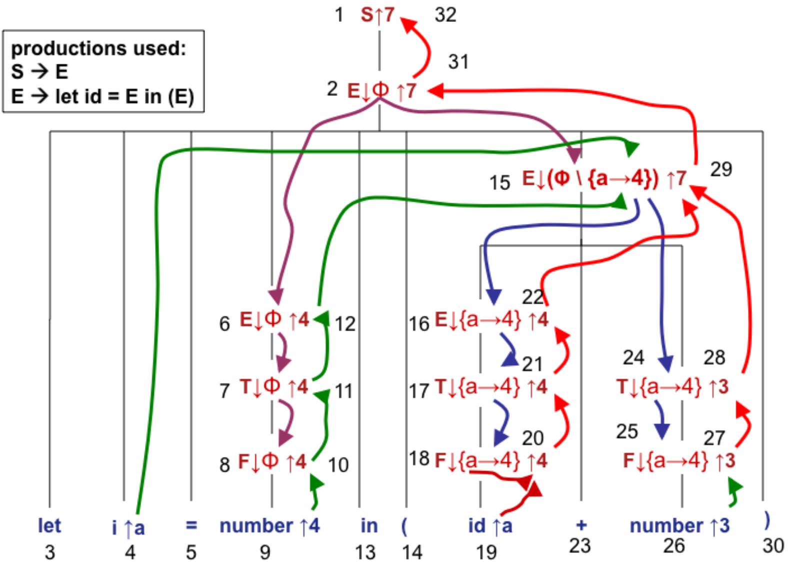
- An AG for associating *type* information with names in variable declarations
- $AI(L) = AI(ID) = \{type\downarrow : \{integer, real\}\}$
  $AS(T) = \{type\uparrow : \{integer, real\}\}$
  $AS(ID) = AS(identifier) = \{name\uparrow : string\}$
  1. $DList \rightarrow D \mid DList\ ;\ D$
  2. $D \rightarrow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

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*

Example of LAG - 2

1. $S \longrightarrow E \; \{E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow\}$
2. $E_1 \longrightarrow E_2 + T \; \{E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow\}$
3. $E \longrightarrow T \; \{T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow\}$
4. $E_1 \longrightarrow let\ id = E_2\ in\ (E_3)$
   $\{E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_3.symtab \downarrow := E_1.symtab \downarrow \backslash \{id.name \uparrow \rightarrow E_2.val \uparrow\}\}$
   Note: changing the above production to:
   $E_1 \rightarrow return\ (E_3)\ with\ id = E_2$ **(with the same computation rules) changes this AG into non-LAG**
5. $T_1 \longrightarrow T_2 * F \; \{T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$
   $T_2.symtab \downarrow := T1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow\}$
6. $T \longrightarrow F \; \{T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow\}$
7. $F \longrightarrow (E) \; \{F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow\}$
8. $F \longrightarrow number \; \{F.val \uparrow := number.val \uparrow\}$
9. $F \longrightarrow id \; \{F.val \uparrow := F.symtab \downarrow [id.name \uparrow]\}$

Example of LAG - 2, 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
- 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

Example 1: SATG for 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 */
     ;
%%
```

Example 2: SATG for 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 */
     ;
%%
```

Example 3: LAG, LATG, and SATG

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.** $D' \rightarrow \epsilon\ |;\ DList$

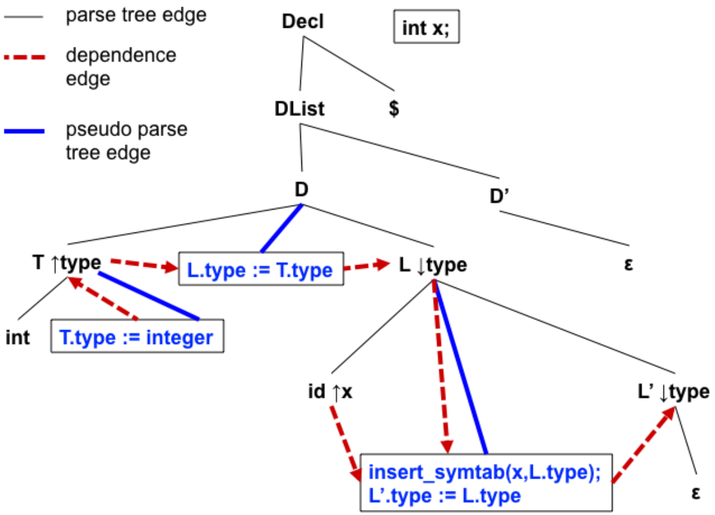
4. $D \rightarrow T\ \{L.type \downarrow := T.type \uparrow\}\ L$

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. $L' \rightarrow \epsilon\ |\ , \{L.type \downarrow := L'.type \downarrow; \}\ L$

Example - 3: LATG Dependence Example



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

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();
}
```

Integrating LATG into RD Parser - 2

```
/* 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 ; }
```

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) }
```

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

Note: array declarations have two possibilities

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

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.