Lecture 06
Semantic Analysis

Outline

- Semantic Analysis
  - Attributes and Attribute Grammars
  - Dependency Graphs and Algorithms for Attribute Computation
  - Symbol Table and Scope Checking
  - Type Checking for Semantic Analysis of a Program

II. Algorithms for Attribute Computation

- 1. Dependency Graphs and Evaluation Order
- 2. Synthesized and Inherited Attributes
  - a Synthesized attributes
  - b Inherited attributes

1 Dependency Graphs and Evaluation Order

- Dependency Graph of a grammar rule
  - Given an attribute grammar, each grammar rule has an associated dependency graph
  - Each attribute $X_i.a_j$ of each symbol corresponds to a node
  - For each attribute equation $X_i.a_j=f_{ij}(\ldots,X_m.a_k,\ldots)$, there is an edge from each node $X_m.a_k$ in the right-hand side to the node $X_i.a_j$ (expressing the dependency of $X_i.a_j$ on $X_m.a_k$)

Example

Attribute grammar for variable declarations

The dependency graph for grammar rule:

- varlist1->id,varlist2 {id.dtype=varlist1.dtype varlist2.dtype=varlist1.dtype}

$$\begin{tikzcd} & \text{varlist1.dtype} \arrow[dl] \arrow[dr] & \\ \text{id.dtype} & & \text{varlist2.dtype} \end{tikzcd}$$

- varlist->id {id.dtype=varlist.dtype}

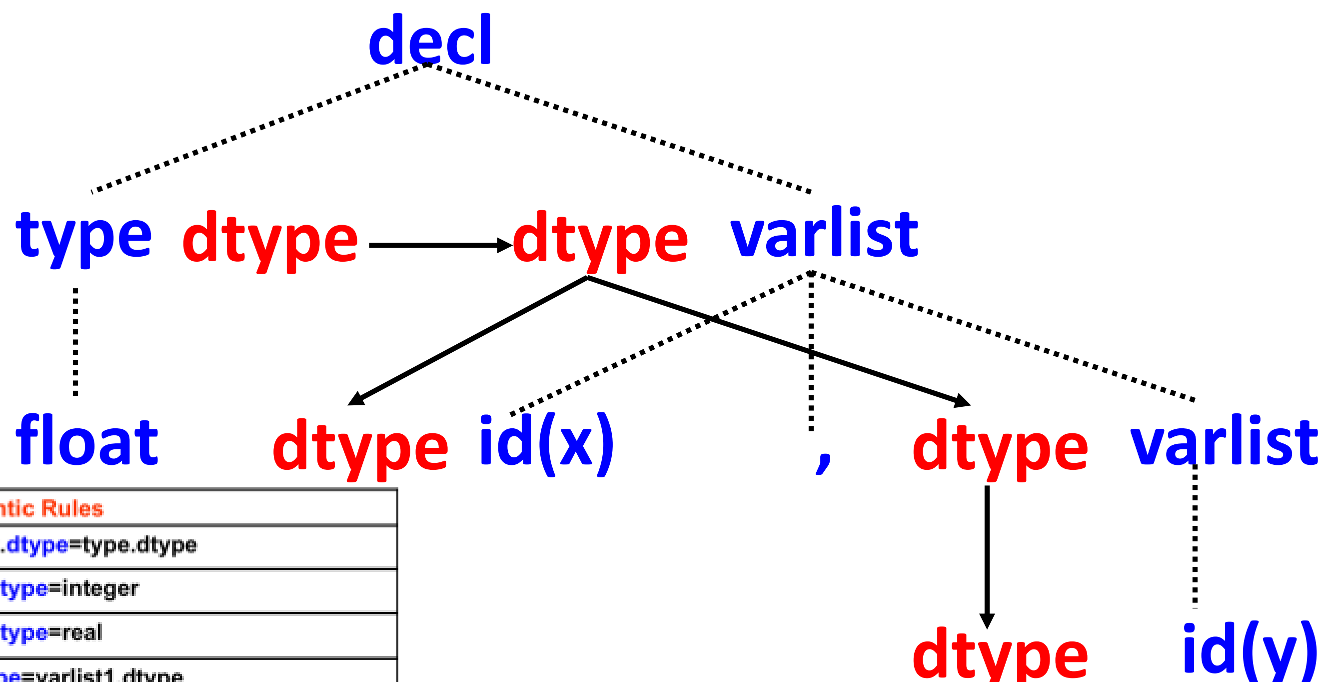
$$\begin{tikzcd} \text{varlist.dtype} \arrow[d] \\ \text{id.dtype} \end{tikzcd}$$

- decl->type varlist {varlist.dtype=type.dtype}

$$\text{type.dtype} \longrightarrow \text{varlist.dtype}$$

Dependency graph of a legal string

- The dependency graph of a legal string generated by the context-free grammar is the union of the dependency graphs of the grammar rule choices representing each node of the parse tree of the string
  - **The dependency graph for string "float x,y"**



Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype=varlist1.dtype
var-list->id	id.dtype=varlist.dtype

2 Synthesized and Inherited Attributes

- Attribute evaluation depends on an explicit or implicit traversal of the parse tree
- Different kinds of traversals vary in power in terms of the kinds of attribute dependencies that can be handled
- We must classify attributes by the kinds of dependencies they exhibit
  - Synthesized attribute
  - Inherited attribute

a Synthesized Attribute

- An attribute is synthesized if all its dependencies point from child to parent in the parse tree.
- Equivalently, an attribute *a* is synthesized if, given each grammar rule A->X1X2…Xn, the only associated attribute equation with an *a* on the left-hand side is of the form

A.a=f(x1.a1,..X1.ak,…,Xn.a1,…Xn.ak)

Example: Synthesized attribute

Attribute grammar for simple integer arithmetic expression

Grammar rule	Semantic Rules
$E \rightarrow E^1 + T$	$E.val = E^1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T^1 * F$	$T.val = T^1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow num$	$F.val = num.val$

The *val* attribute is synthesized

Evaluation of Synthesized Attributes

- Given that a parse tree or syntax tree has been constructed by a parser
- The synthesized attribute values can be computed by a single bottom-up, or postorder traversal of the tree
- Express this by the following code:

```
procedure PostEval(T:treenode)
begin
    for each child C of T do
        PostEval(C);
    compute all synthesized attributes of T;
end;
```

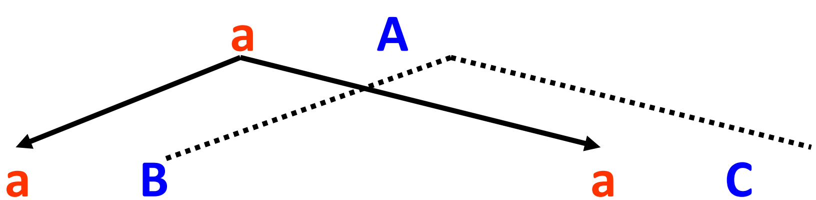
S-attributed grammar

- An attribute grammar in which all attributes are synthesized is called an S-attributed grammar

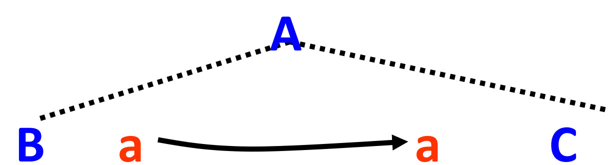
b Inherited Attribute

- An attribute that is not synthesized is called an inherited attribute
- Inherited attributes have dependencies that flow either from parent to children in the parse tree or from sibling to sibling

Two basic kinds of dependency of inherited attributes:



(a) Inheritance from parent to siblings



(b) Inheritance from sibling to sibling

Example: Inherited Attribute

Attribute grammar for variable declarations

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype =varlist1.dtype
var-list->id	id.dtype=varlist.dtype

The *dtype* attribute is inherited

Evaluation of Inherited Attributes

- Inherited attributes can be computed by a preorder traversal of the parse tree

- Express this by the following code

```
procedure PreEval(T:treenode);
begin
    for each child C of T do
        compute all inherited attributes of C;
        PreEval(C);
end;
```

- Unlike synthesized attributes, since inherited attributes may have dependencies among the attributes of the children, the order in which the inherited attributes of the children are computed is important

Attribute Computation During Parsing

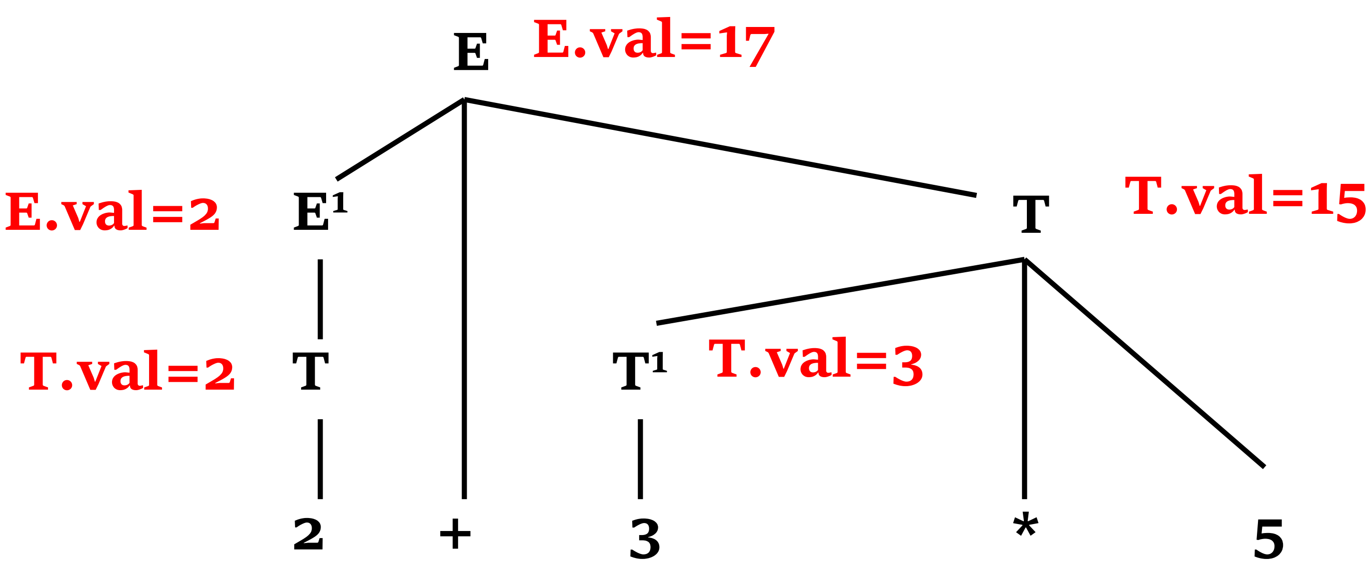
- Attributes can be computed at the same time as the parsing stage, without waiting to perform further passes over the source code by recursive traversals of the syntax tree

Example

attribute grammar for arithmetic expression:

1) $E \to E^1 + T$ { $E.val := E^1.val + T.val$ }
2) $E \to T$ { $E.val := T.val$ }
3) $T \to T^1 * number$ { $T.val := T^1.val * number.val$ }
4) $T \to number$ { $T.val := number.val$ }

The process of bottom-up parsing and the parse tree for “2+3*5”



When the parsing completes, the attribute value is also computed

Attribute Computation During Parsing

- Which Attributes can be Computed During the Parse?
  - It depends on the power and properties of the parsing method employed.
    - All the major parsing methods process the input program form left to right
    - This is equivalent to the requirement that the attributes be capable of evaluation by a left-to-right traversal of the parse tree
    - For synthesized attributes this is not a restriction, since the children of a node can be processed in arbitrary order

Attribute Computation During Parsing

- Which Attributes can be Computed During the Parse?
  - It depends on the power and properties of the parsing method employed.(continued)
    - But, for inherited attributes, this means that there may be no "backward" dependencies (dependencies pointing from right to left in the parse tree) in the dependency graph
    - Attribute grammars that do satisfy this property are called L-attributed (Left to right)

L-attributed Grammar

- An attribute grammar for attributes is L-attributed if,
  - each attribute is synthesized, or
  - for each inherited attribute a_j and each grammar rule

$$X_o \rightarrow X_1 X_2 \ldots X_{i-1} X_i \ldots X_n$$

    - the associated equations for a_j are all of the form

$$X_i.a_j = f_{ij}(X_o.a_1, \ldots, X_o.a_k, X_1.a_1, \ldots X_1.a_k, \ldots X_{i-1}.a_1, \ldots X_{i-1}.a_k)$$

    - here, the value of a_j at X_i can only depend on attributes of the symbols $X_o, \ldots, X_{i-1}$ that occur to the left of X_i
    - there are no cycles in a dependency graph formed by the attributes of X_i

L-attributed Grammar

- In L-attributed grammar, the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left.

- As a special case, an S-attributed grammar is L-attributed