Chapter 7: Syntax-Directed Translation

陳奇業 成功大學資訊工程系

Semantic Analyzer

- Semantic Structure
  - What is the program supposed to do?
  - Semantics analysis can be done during syntax analysis phase or intermediate code generator phase or the final code generator.
  - typical static semantic features include declarations and type checking.
  - information (attributes) gathered can be either added to the tree as annotations or entered into the symbol table.

Two Categories of Semantic Analysis

- Semantic analysis can be divided into two categories.
  - The first is the analysis of a program required by the rules of the programming language to establish its correctness and to guarantee proper execution.
  - The second category of semantic analysis is the analysis performed by a compiler to enhance the efficiency of execution of the translated program. This kind of analysis is usually included in discussions of optimization, or code improving techniques.

Compiling Process & Compiler Structure

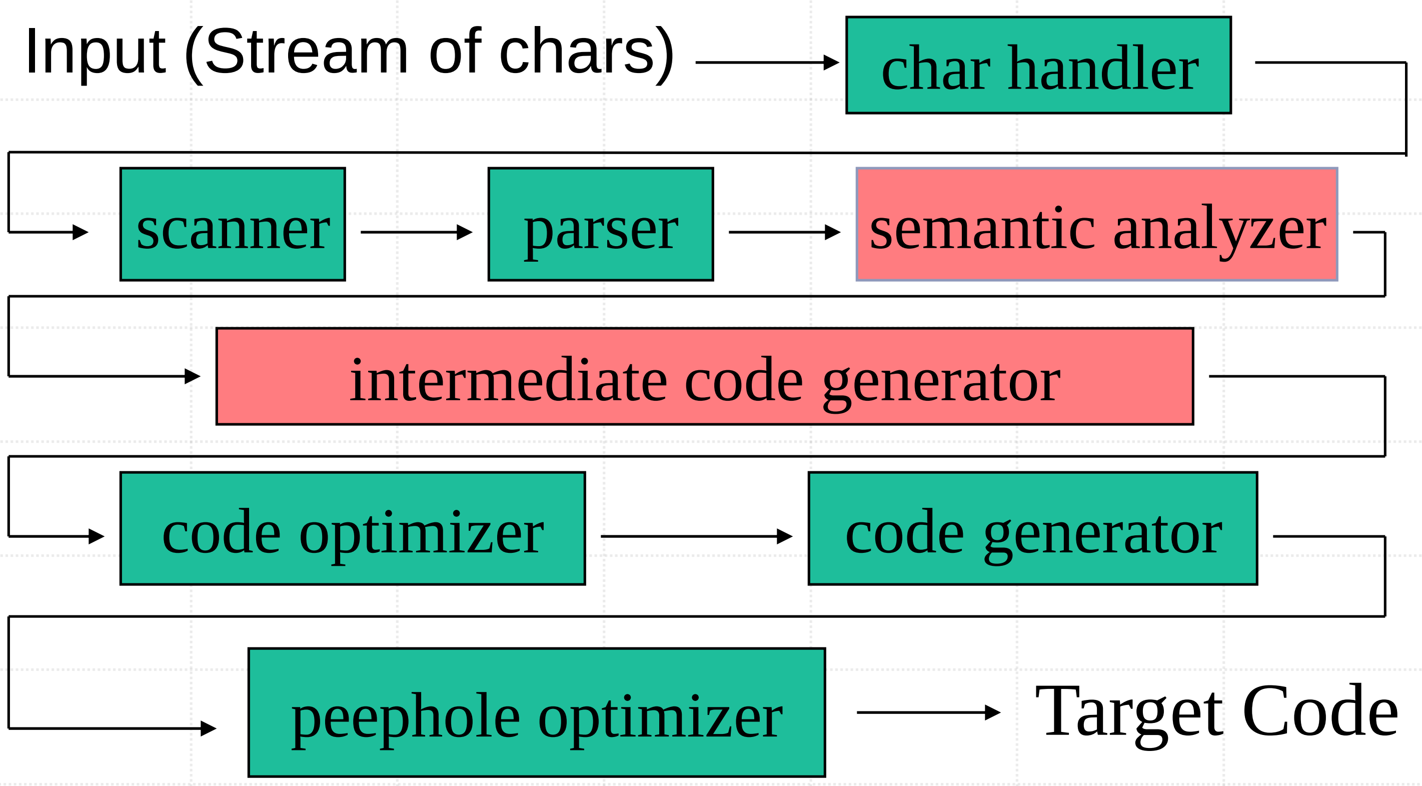
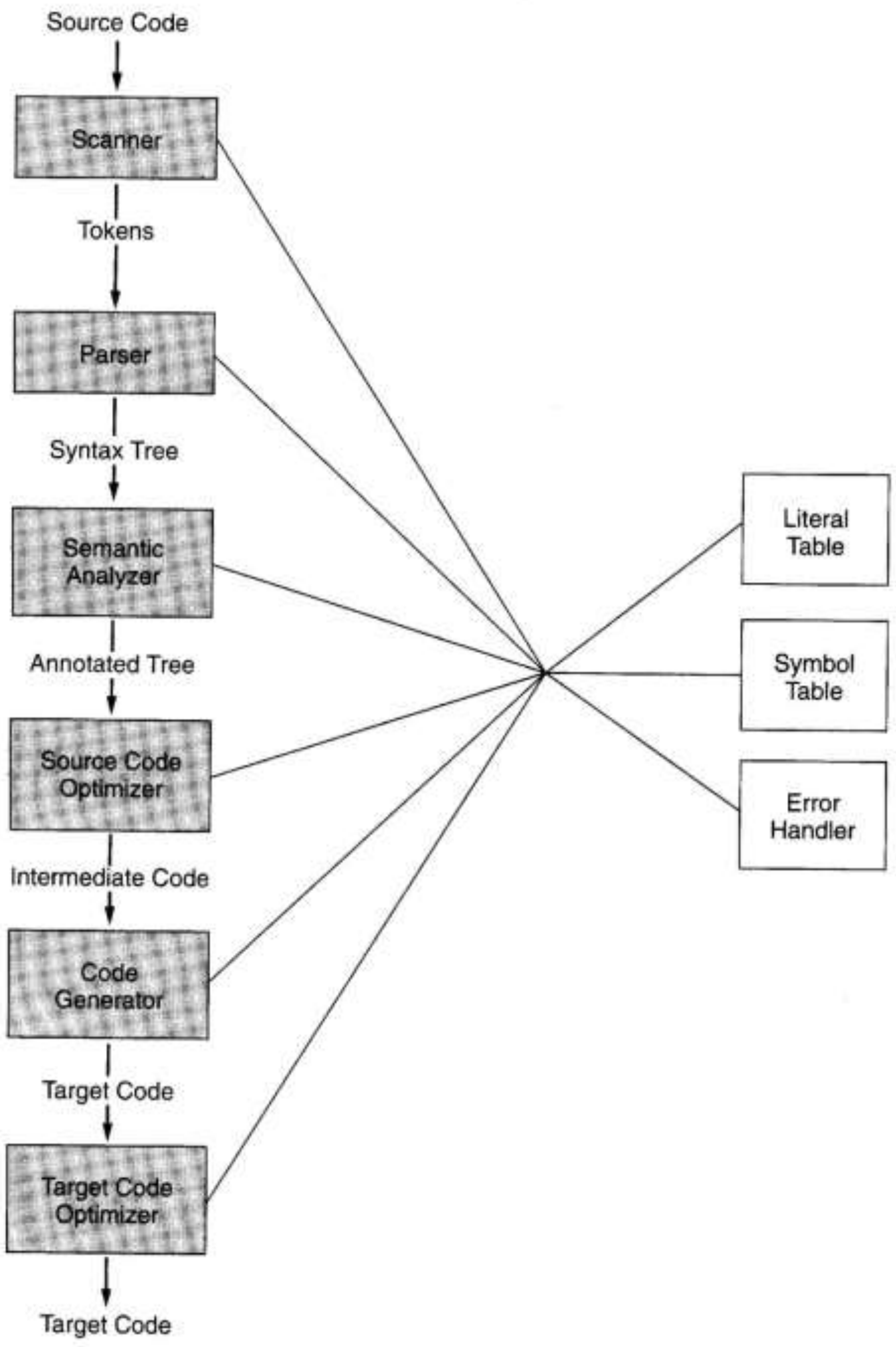
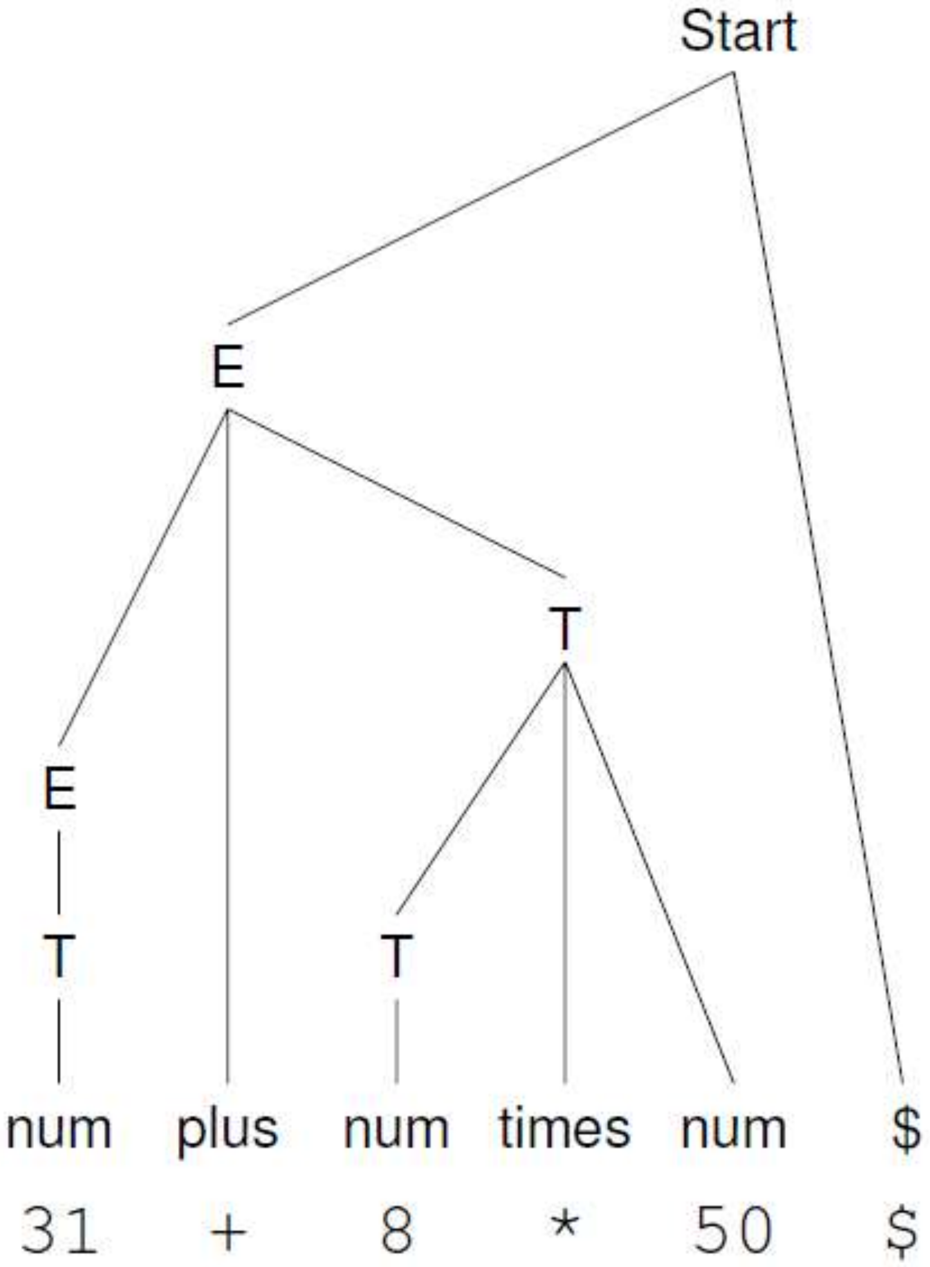
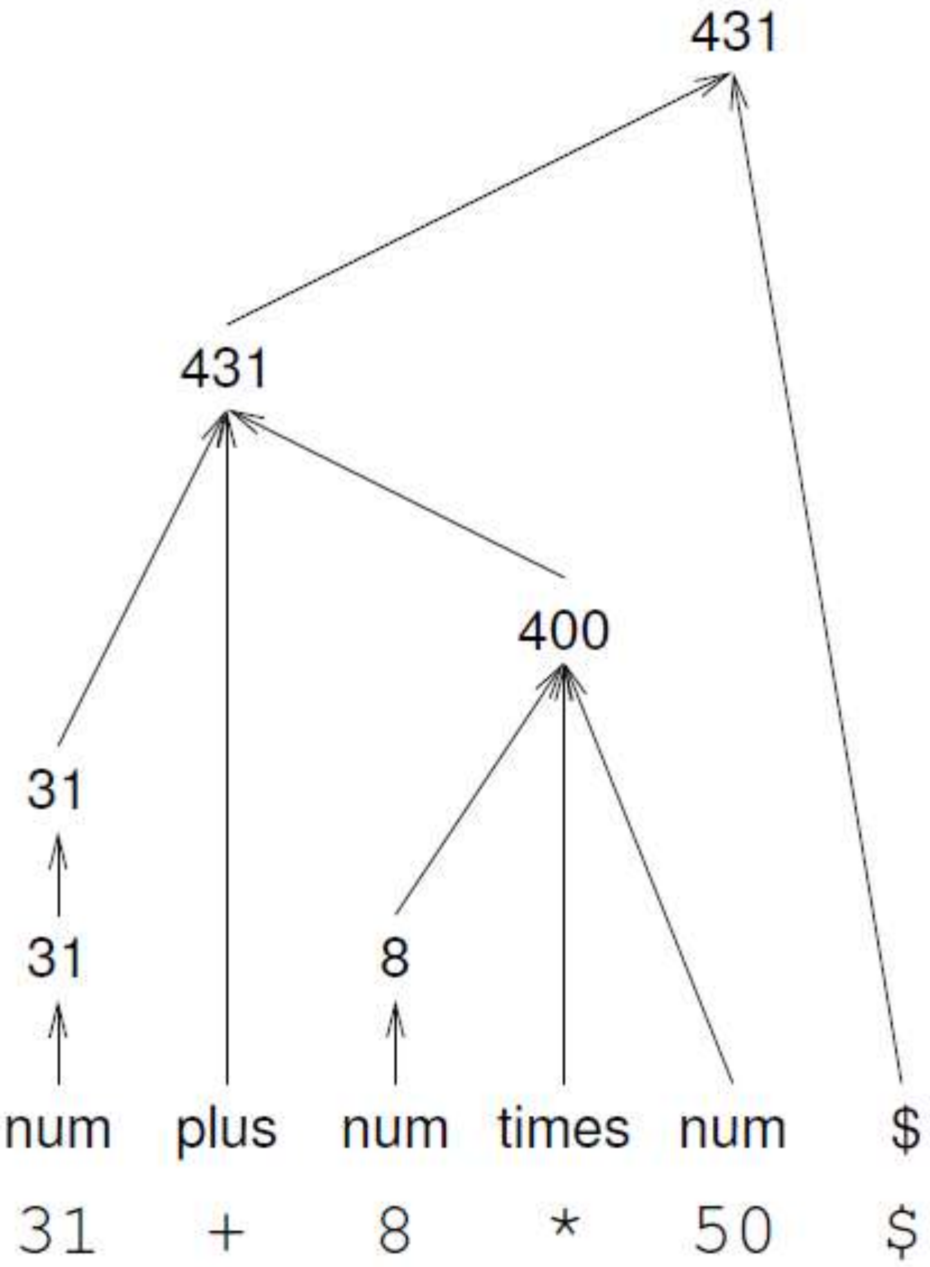


Figure 1.1
The phases of a compiler





Syntax



Semantic Values

Semantic Analysis Process

- includes formally:
  - description of the analyses to perform
  - implementation of the analysis (translation of the description) that may use appropriate algorithms.

Description of Semantic Analysis

1. Identify attributes (properties) of language (syntactic) entities.

2. Write attribute equations (or semantic rules) that express how the computation of such attributes is related to the grammar rules of the language.

Such a set of attributes and equations is called an attribute grammar.

Syntax-directed semantics

- The semantic content of a program is closely related to its syntax.

- All modern languages have this property.

Attributes

- An attribute is any property of a programming language construct.
- Typical examples of attributes include the data type of a variable, the value of an expression, the location of a variable in memory, the object code of a procedure, the number of significant digits in a number.
- Attribute corresponds to the name of a field of a structure.

Attribute Grammars

- In syntax-directed semantics, attributes are associated with grammar symbols of the language. That is, if X is a grammar symbol and a is an attribute associated to X, then we write $X.a$ for the value of an associated to X.

- For each grammar rule $X_0 \rightarrow X_1 X_2 \cdots X_n$ the values of the attributes $X_i.a_j$ of each grammar symbol X_i are related to the values of the attributes of other grammar symbols in the rule.

Attribute Grammars

- That is, each relationship is specified by an attribute equation or semantic rule of the form:

$$X_i.a_j = f_{ij}(X_0.a_1, \dots, X_0.a_k, X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

- An attribute grammar for the attributes $a_1, \dots, a_k$ is the collection of all such attribute equations (semantic rules), for all the grammar rules of the language.

Example 6.2 Consider the following grammar for simple integer arithmetic expressions:

$$exp \rightarrow exp + term \mid exp - term \mid term$$
$$term \rightarrow term * factor \mid factor$$
$$factor \rightarrow (\ exp\ ) \mid \texttt{number}$$

This grammar is a slightly modified version of the simple expression grammar studied extensively in previous chapters. The principal attribute of an *exp* (or *term* or *factor*) is its numeric value, which we write as *val*. The attribute equations for the *val* attribute are given in Table 6.2.

Table 6.2

Attribute grammar for Example 6.2

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (\ exp\ )$	$factor.val = exp.val$
$factor \rightarrow \texttt{number}$	$factor.val = \texttt{number}.val$

number.val must be computed prior to factor.val

These equations express the relationship between the syntax of the expressions and the semantics of the arithmetic computations to be performed. Note, for example, the difference between the syntactic symbol + (a token) in the grammar rule

$$exp_1 \rightarrow exp_2 + term$$

and the arithmetic addition operation + to be performed in the equation

$$exp_1.val = exp_2.val + term.val$$

Attribute grammars may involve several interdependent attributes.

Attribute grammar for octal or decimal number indicated by one-character suffix o (for octal) or d (for decimal)

Table 6.4
Attribute grammar for Example 6.4

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

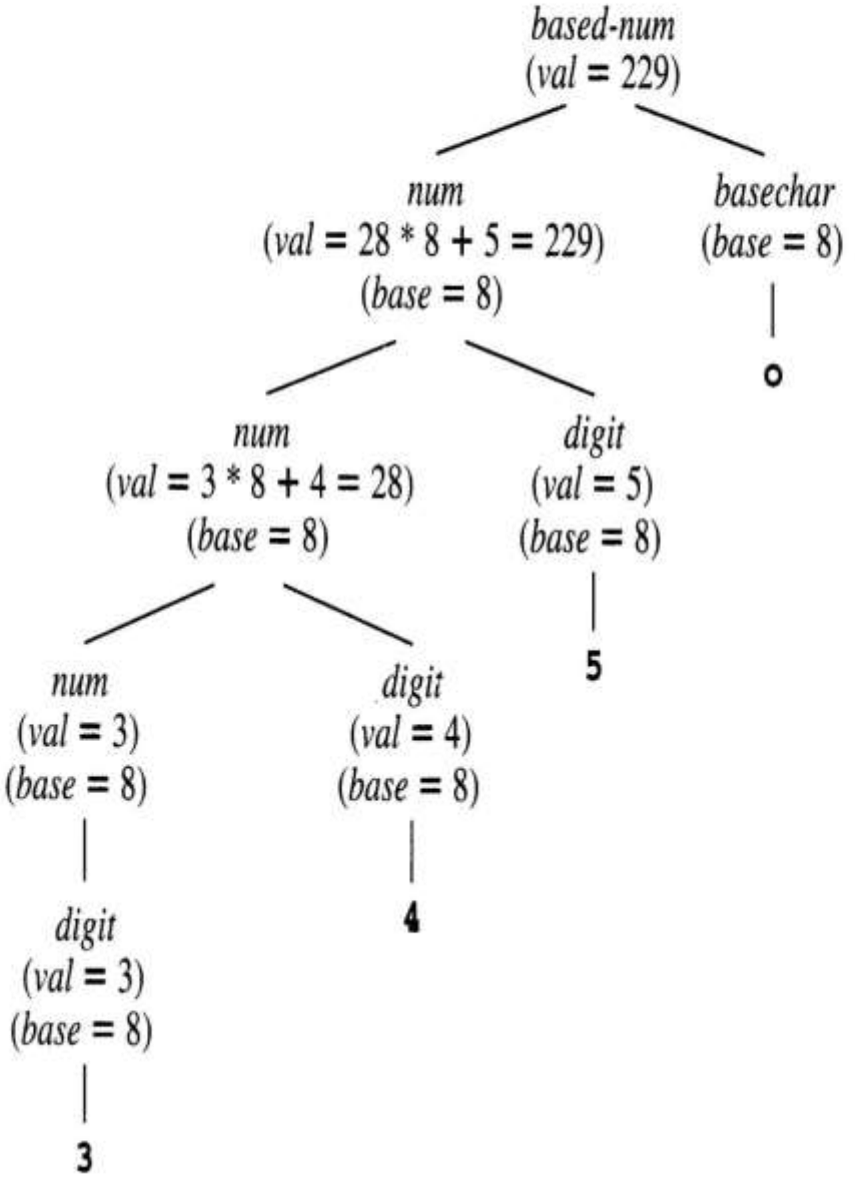
e.g. 345o

128d

128o (x)

Figure 6.4

Parse tree showing attribute computations for Example 6.4



§

Attribute grammars may be defined for different purposes.

Table 6.6

Attribute grammar for abstract syntax trees of simple integer arithmetic expressions

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** *term*	exp_1 .*tree* = *mkOpNode* (**+**, exp_2 .*tree*, *term.tree*)
$exp_1 \rightarrow exp_2$ **-** *term*	exp_1 .*tree* = *mkOpNode*(**-**, exp_2 .*tree*, *term.tree*)
$exp \rightarrow term$	*exp.tree* = *term.tree*
$term_1 \rightarrow term_2$ ***** *factor*	$term_1$.*tree* = *mkOpNode*(*****, $term_2$.*tree*, *factor.tree*)
$term \rightarrow factor$	*term.tree* = *factor.tree*
$factor \rightarrow$ **(** *exp* **)**	*factor.tree* = *exp.tree*
$factor \rightarrow$ **`number`**	*factor.tree* = *mkNumNode*(**`number`**.*lexval*)

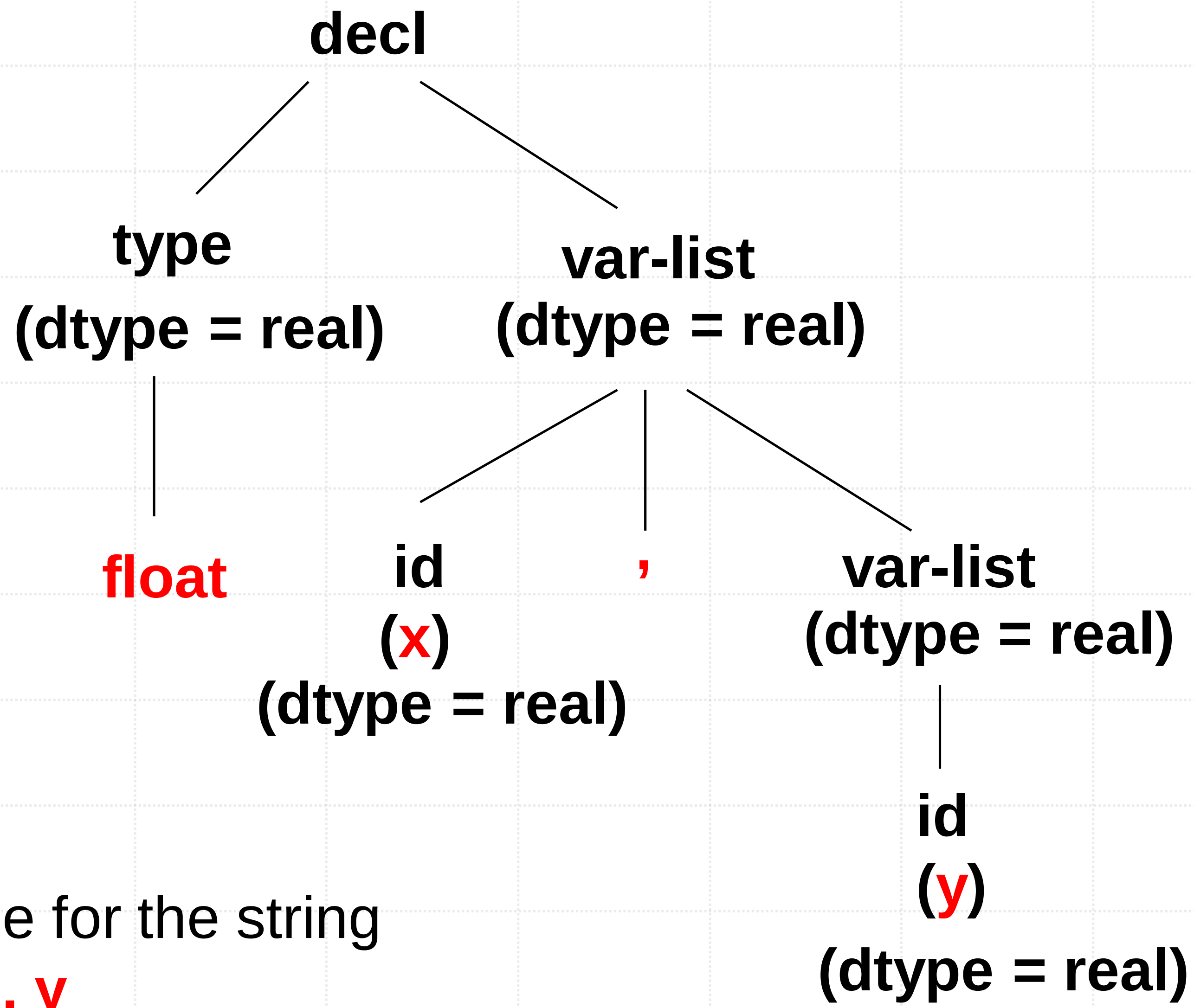
§

Algorithms for attribute computation

- Dependency graph and evaluation order

Attribute grammar for simple C-like variable declarations

Grammar Rules	Semantic Rules
decl → type var-list	var-list.dtype = type.dtype
type → *int*	type.dtype = integer
type → *float*	type.dtype = real
var-list_1 → *id* , var-list_2	id.dtype = var-list_1.dtype var-list_2.dtype = var-list_1.dtype
var-list → *id*	id.dtype = var-list.dtype



Parse tree for the string
float x , y

Example 6.7 Consider the grammar of Example 6.3, with the attribute grammar for the attribute *dtype* given in Table 6.3. In this example the grammar rule $\textit{var-list}_1 \rightarrow \textbf{id}\textbf{,}\ \textit{var-list}_2$ has the two associated attribute equations

$$
\begin{aligned}
&\textbf{id}.\,\textit{dtype} = \textit{var-list}_1.\textit{dtype} \\
&\textit{var-list}_2.\textit{dtype} = \textit{var-list}_1.\textit{dtype}
\end{aligned}
$$

and the dependency graph

$$
\begin{tikzcd}
& \textit{var-list.dtype} \arrow[dl] \arrow[dr] & \\
\textbf{id}.\textit{dtype} & & \textit{var-list.dtype}
\end{tikzcd}
$$

Similarly, the grammar rule $\textit{var-list} \rightarrow \textbf{id}$ has the dependency graph

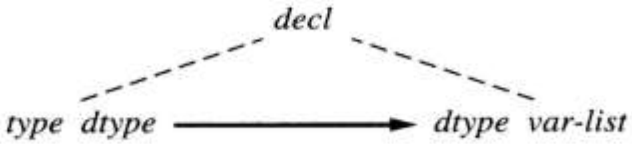
$$
\begin{array}{c}
\textit{var-list.dtype} \\
\downarrow \\
\textbf{id}\,.\textit{dtype}
\end{array}
$$

The two rules $\textit{type} \rightarrow \textbf{int}$ and $\textit{type} \rightarrow \textbf{float}$ have trivial dependency graphs.

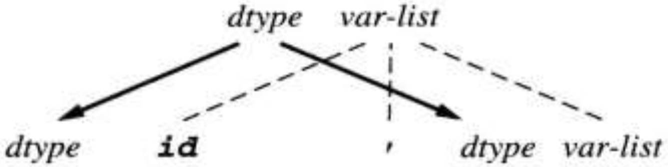
Finally, the rule $\textit{decl} \rightarrow \textit{type var-list}$ with the associated equation $\textit{var-list.dtype} = \textit{type.dtype}$ has the dependency graph

$$
\textit{type.dtype} \quad \longrightarrow \quad \textit{var-list.dtype}
$$

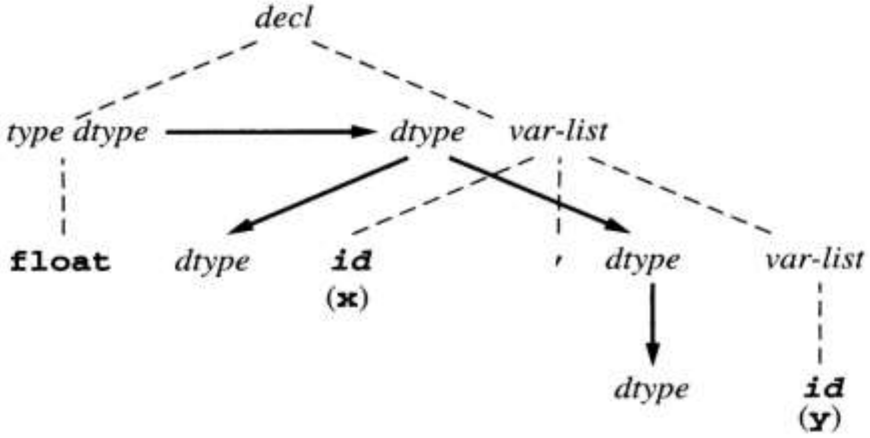
In this case, since *decl* is not directly involved in the dependency graph, it is not completely clear which grammar rule has this graph associated to it. For this reason (and a few other reasons that we discuss later), we often draw the dependency graph superimposed over a parse tree segment corresponding to the grammar rule. Thus, the above dependency graph can be drawn as



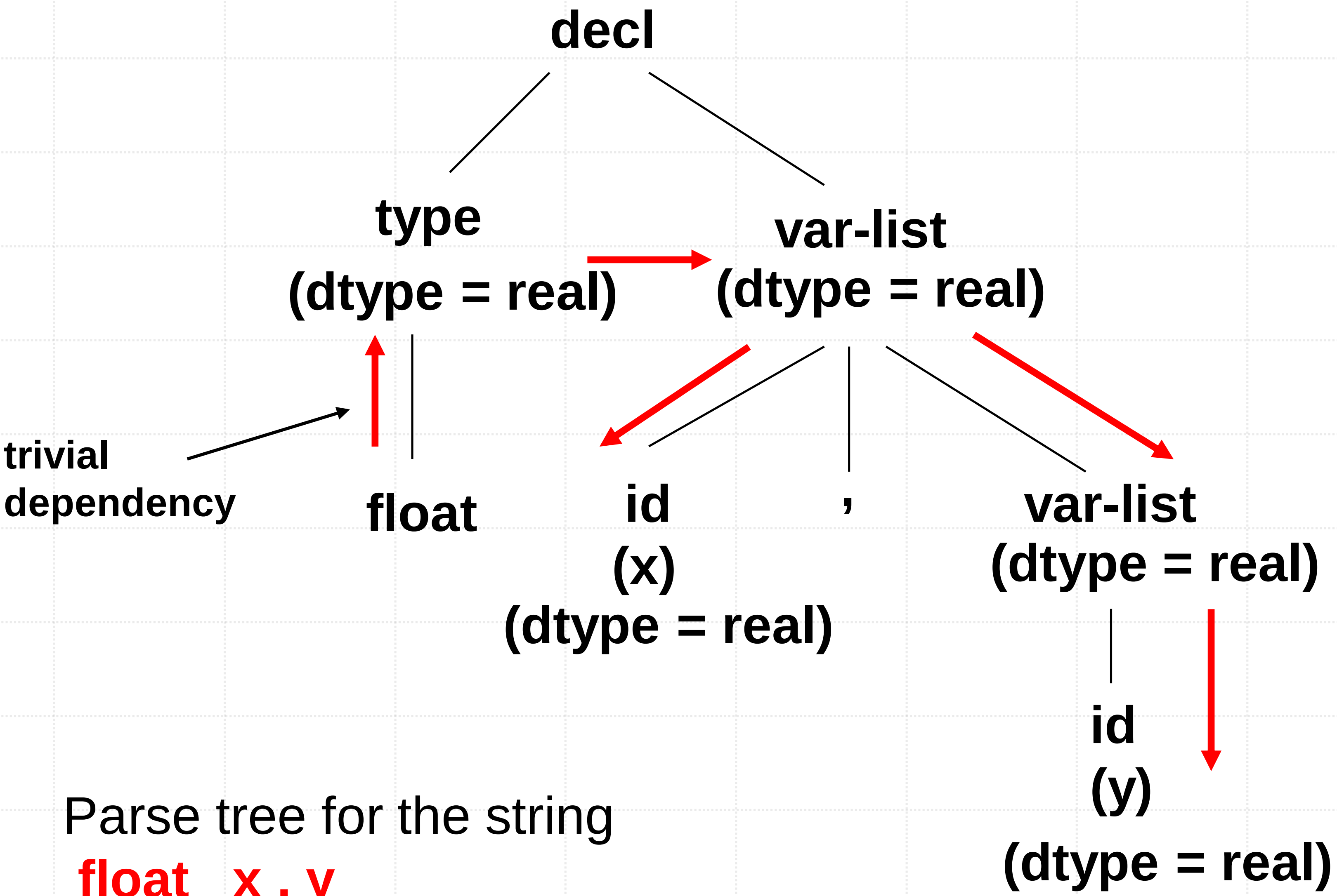
and this makes clearer the grammar rule to which the dependency is associated. Note, too, that when we draw the parse tree nodes we suppress the dot notation for the attributes, and represent the attributes of each node by writing them next to their associated node. Thus, the first dependency graph in this example can also be written as



Finally, the dependency graph for the string **`float x,y`** is



§



Parse tree for the string
float x , y

Then the pseudocode for a recursive procedure that computes the *dtype* attribute at all required nodes is as follows:

```
procedure EvalType ( T: treenode );
begin
  case nodekind of T of
  decl:
        EvalType ( type child of T );
        Assign dtype of type child of T to var-list child of T;
        EvalType ( var-list child of T );
  type:
        if child of T = int then T.dtype := integer
        else T.dtype := real;
  var-list:
        assign T.dtype to first child of T;
        if third child of T is not nil then
           assign T.dtype to third child;
           EvalType ( third child of T );
  end case;
end EvalType;
```

```
Procedure PreEval (T: treenode);

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

Algorithm for evaluating inherited attributes ➔ preorder traversal

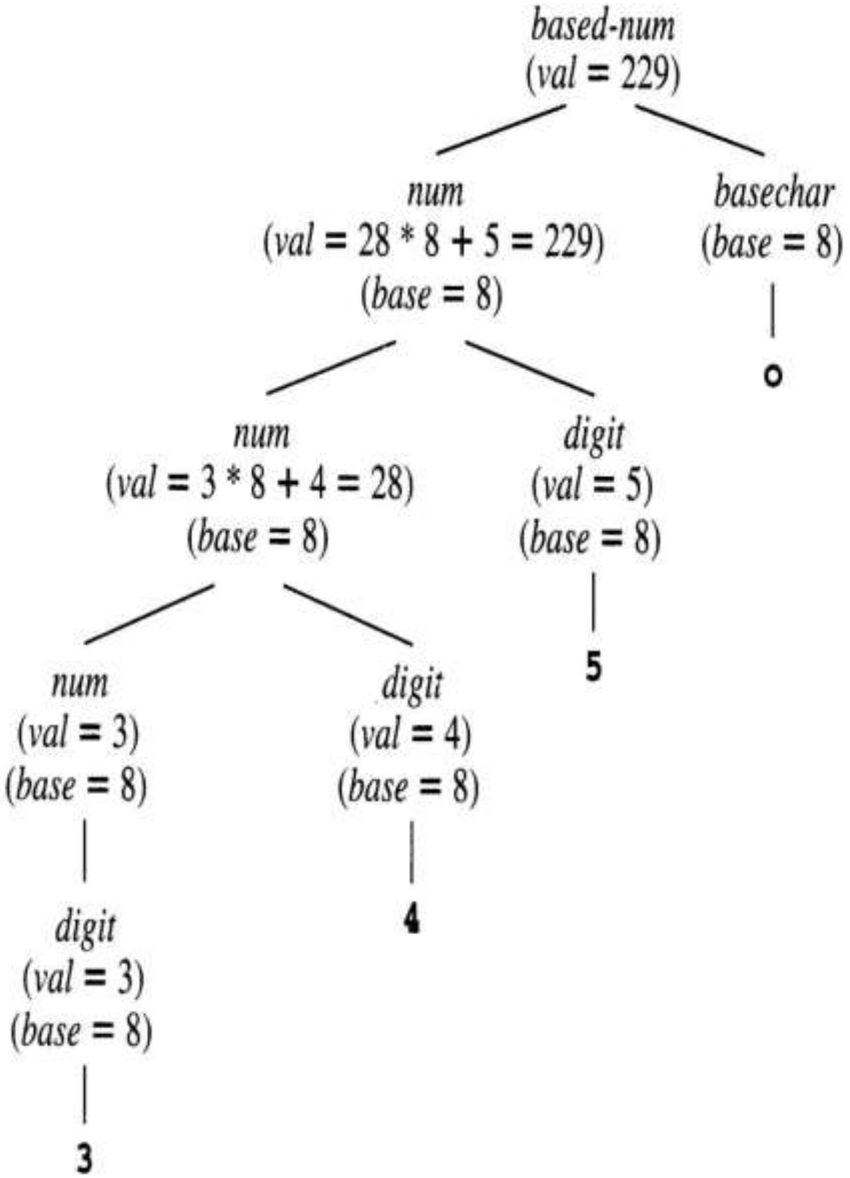
Attribute grammar for octal or decimal number indicated by one-character suffix o (for octal) or d (for decimal)

Table 6.4
Attribute grammar for Example 6.4

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2$ *digit*	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

Figure 6.4

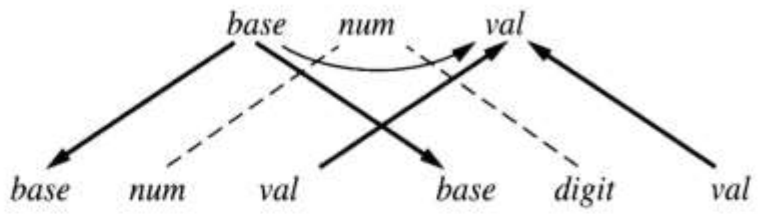
Parse tree showing attribute computations for Example 6.4



§

This graph expresses the dependencies of the two associated equations *based-num.val* = *num.val* and *num.base* = *basechar.base*.

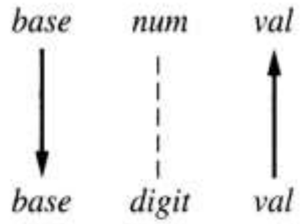
Next we draw the dependency graph corresponding to the grammar rule *num* → *num digit*:



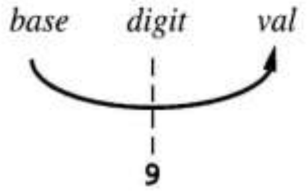
This graph expresses the dependencies of the three attribute equations

$$
\begin{aligned}
& num_1\,.val = \\
& \qquad \textbf{if } digit.val = error \textbf{ or } num_2\,.val = error \\
& \qquad \textbf{then } error \\
& \qquad \textbf{else } num_2\,.val * num_1\,.base + digit.val \\
& num_2\,.base = num_1\,.base \\
& digit.base = num_1\,.base
\end{aligned}
$$

The dependency graph for the grammar rule *num* → *digit* is similar:

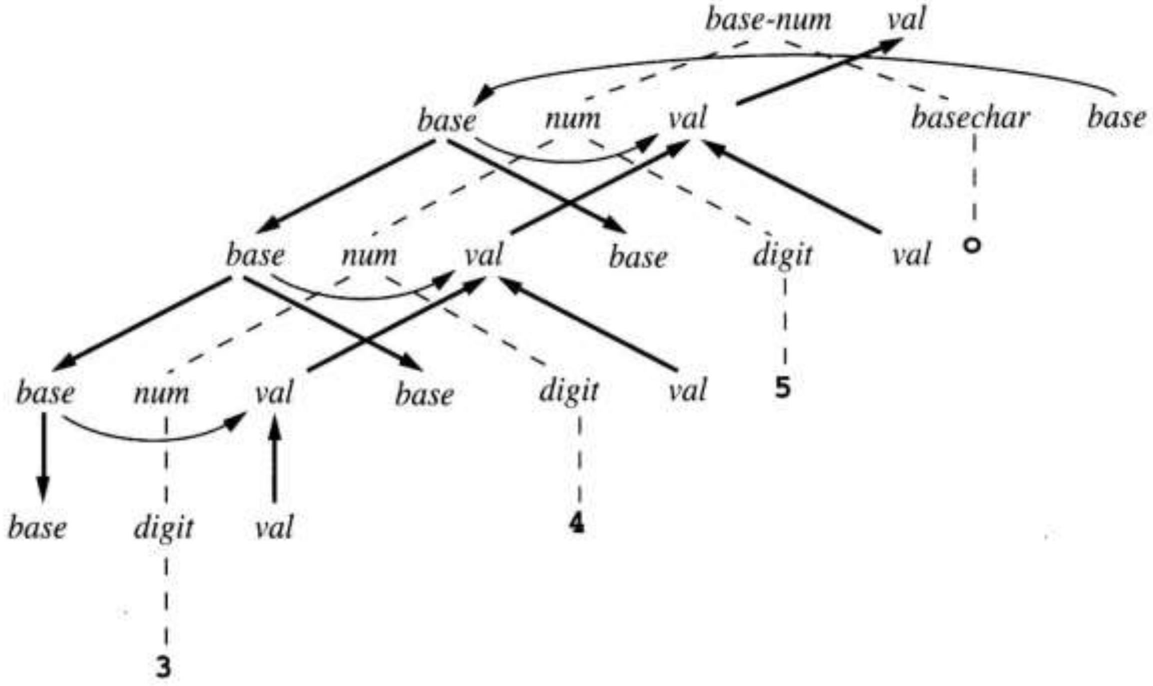


Finally, we draw the dependency graph for the grammar rule *digit* → **9**:



This graph expresses the dependency created by the equation *digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9, namely, that *digit.val* depends on *digit.base* (it is part of the test in the **if**-expression). It remains to draw the dependency graph for the string **345o**. This is done in Figure 6.6.

Figure 6.6
Dependency graph for the string **345o** (Example 6.8)



§

Example 6.9 The dependency graph of Figure 6.6 is a DAG. In Figure 6.7 we number the nodes of the graph (and erase the underlying parse tree for ease of visualization). One topological sort is given by the node order in which the nodes are numbered. Another topological sort is given by the order

12 6 9 1 2 11 3 8 4 5 7 10 13 14

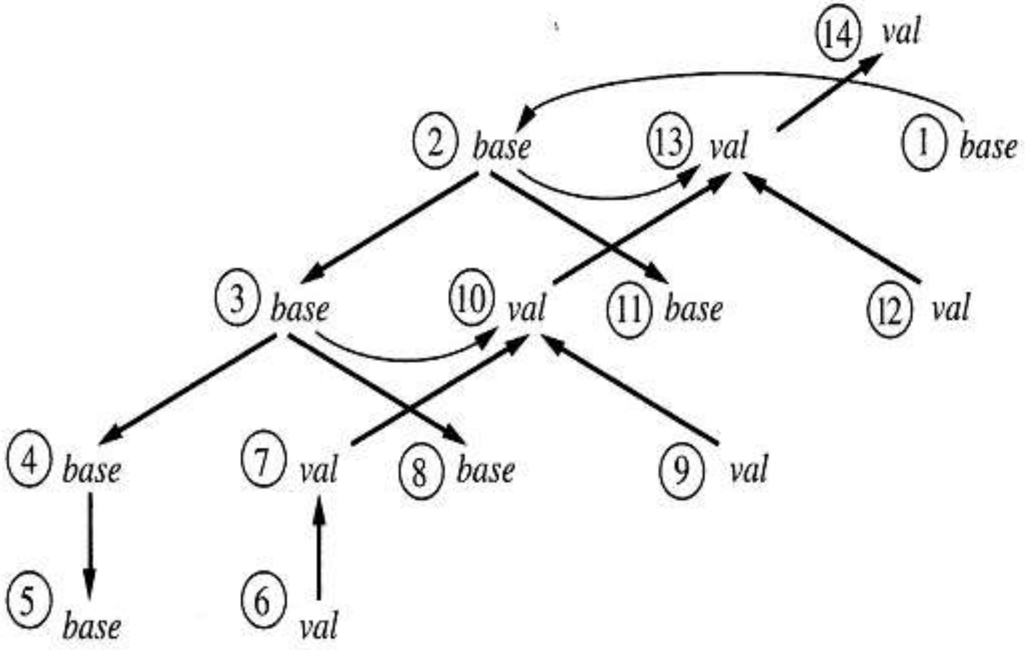


Figure 6.7
Dependency graph for the string **345o** (Example 6.9)

§

```
procedure EvalWithBase ( T: treenode );
begin
  case nodekind of T of
  based-num:
        EvalWithBase ( right child of T );
        assign base of right child of T to base of left child;
        EvalWithBase ( left child of T );
        assign val of left child of T to T.val;
  num:
        assign T.base to base of left child of T;
        EvalWithBase ( left child of T );
        if right child of T is not nil then
           assign T.base to base of right child of T;
           EvalWithBase ( right child of T );
           if vals of left and right children ≠ error then
                T.val := T.base*(val of left child) + val of right child
           else T.val := error;
        else T.val := val of left child;
  basechar:
        if child of T = o then T.base := 8
        else T.base := 10;
  digit:
        if T.base = 8 and child of T = 8 or 9 then T.val := error
        else T.val := numval ( child of T );
  end case;
end EvalWithBase;
```

base is computed in preorder and val in postorder

Synthesized Attributes

- An attribute a is synthesized if, given a grammar rule $A \to X_1 X_2 \cdots X_n$, the only associated attribute equation with an a on the left-hand side is of the form:

$$A.a = f(X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

e.g., $E_1 \to E_2 + E_3 \quad \{E_1.val = E_2.val + E_3.val;\}$

where $E.val$ represents the attribute (numerical value obtained) for E

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

Example 6.2 Consider the following grammar for simple integer arithmetic expressions:

$$
\begin{aligned}
&exp \rightarrow exp \;\texttt{+}\; term \mid exp \;\texttt{-}\; term \mid term \\
&term \rightarrow term \;\texttt{*}\; factor \mid factor \\
&factor \rightarrow \texttt{(}\; exp \;\texttt{)} \mid \textbf{number}
\end{aligned}
$$

This grammar is a slightly modified version of the simple expression grammar studied extensively in previous chapters. The principal attribute of an *exp* (or *term* or *factor*) is its numeric value, which we write as *val*. The attribute equations for the *val* attribute are given in Table 6.2.

Table 6.2

Attribute grammar for Example 6.2

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(** exp **)**	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**.val

These equations express the relationship between the syntax of the expressions and the semantics of the arithmetic computations to be performed. Note, for example, the difference between the syntactic symbol **+** (a token) in the grammar rule

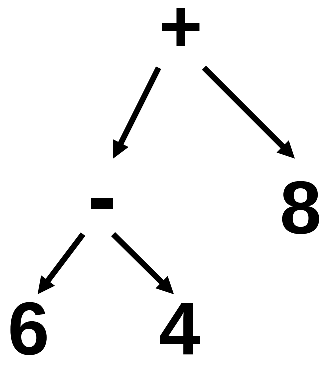
$$exp_1 \rightarrow exp_2 \;\texttt{+}\; term$$

and the arithmetic addition operation + to be performed in the equation

$$exp_1.val = exp_2.val + term.val$$

Figure 6.8
C code for the postorder attribute evaluator for Example 6.11

```c
void postEval(SyntaxTree t)
{ int temp;
  if (t->kind == OpKind)
  { postEval(t->lchild);
    postEval(t->rchild);
    switch (t->op)
    { case Plus:
        t->val = t->lchild->val + t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val - t->rchild->val;
        break;
      case Times:
        t->val = t->lchild->val * t->rchild->val;
        break;
    } /* end switch */
  } /* end if */
} /* end postEval */
```

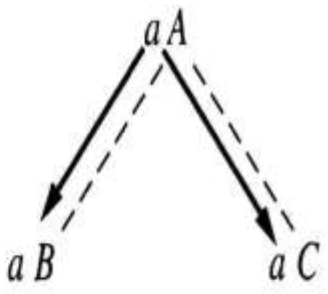


§

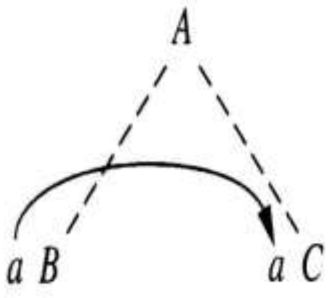
Inherited Attributes

- An attribute that is not synthesized is called an inherited attribute.

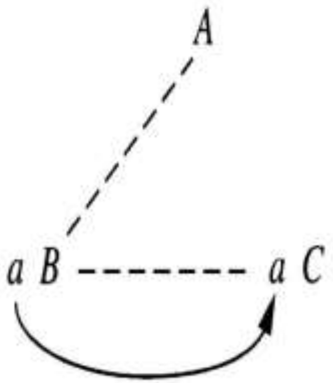
Figure 6.9
Different kinds of inherited dependencies



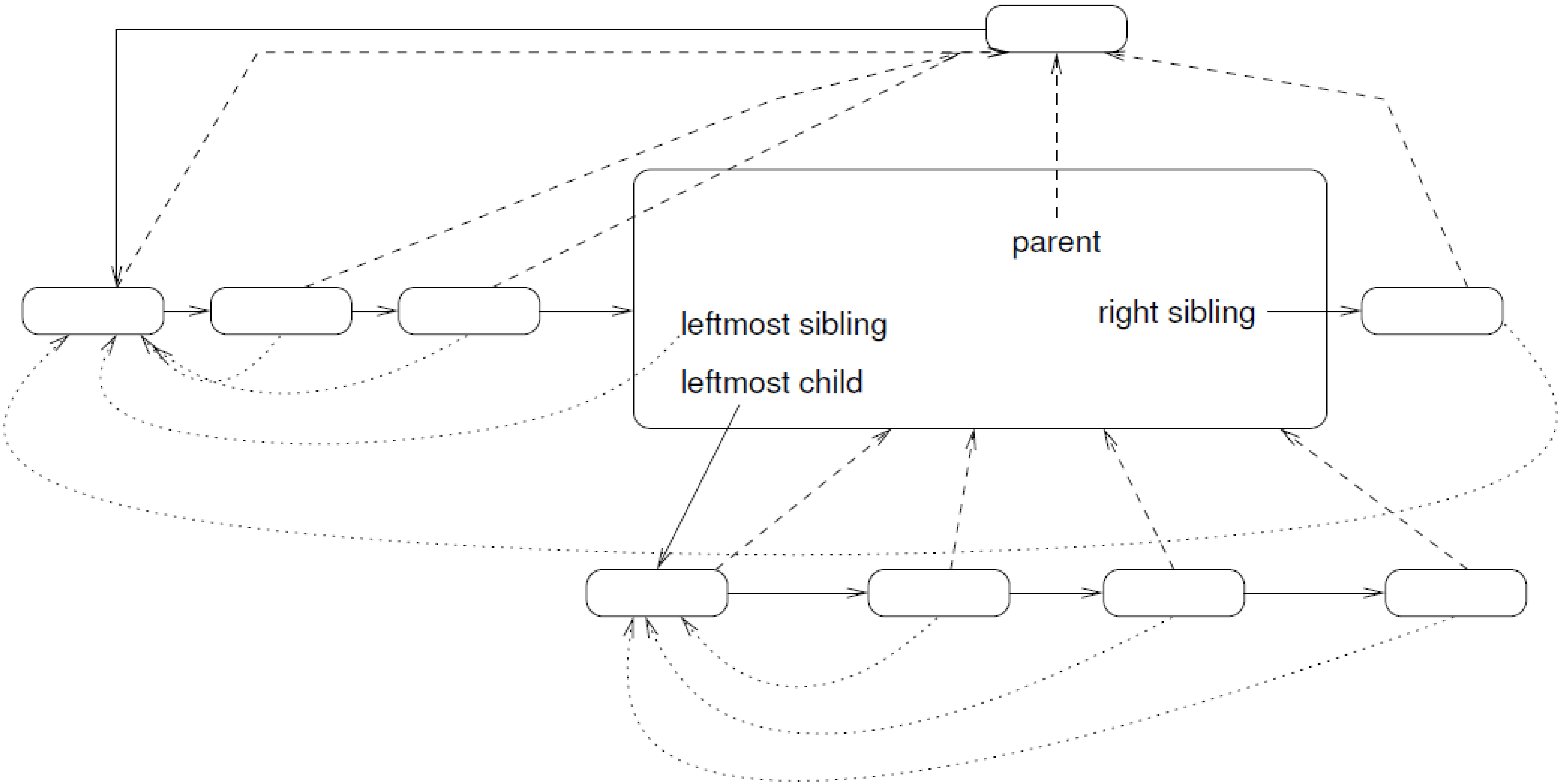
(a) Inheritance from parent to siblings



(b) Inheritance from sibling to sibling



(c) Sibling inheritance via sibling pointers


parent
leftmost sibling
right sibling
leftmost child

Computation of Attributes During Parsing

- L-attributed grammars

Definition

An attribute grammar for attributes $a_1, \ldots, a_k$ is **L-attributed** if, for each inherited attribute a_j and each grammar rule

$$X_0 \rightarrow X_1 X_2 \ldots X_n$$

the associated equations for a_j are all of the form

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$$

That is, the value of a_j at X_i can only depend on attributes of the symbols $X_0, \ldots, X_{i-1}$ that occur to the left of X_i in the grammar rule.

Computing Synthesized Attributes During LR Parsing

- LALR(1) parser are primarily suited to handling synthesized attributes.

- Two stacks are required.
  - value stack and parsing stack

Table 6.8

Parsing and semantic actions for the expression **3*4+5** during an LR parse

	Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
1	$	**3*4+5** $	shift	$	
2	$ **n**	***4+5** $	reduce $E \rightarrow$ **n**	$ **n**	$E.val$ = **n**$.val$
3	$ E	***4+5** $	shift	$ 3	
4	$ E *	**4+5** $	shift	$ 3 *	
5	$ E * **n**	**+5** $	reduce $E \rightarrow$ **n**	$ 3 * **n**	$E.val$ = **n**$.val$
6	$ E * E	**+5** $	reduce $E \rightarrow E * E$	$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	$ E	**+5** $	shift	$ 12	
8	$ E +	**5** $	shift	$ 12 +	
9	$ E + **n**	$	reduce $E \rightarrow$ **n**	$ 12 + **n**	$E.val$ = **n**$.val$
10	$ E + E	$	reduce $E \rightarrow E + E$	$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	$ E	$		$ 17	

Theorem

(From Knuth [1968]). Given an attribute grammar, all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.

Translation (Attribute Computation)

- A translation scheme is merely a context-free grammar in which a program fragment called semantic action is associated with each production.

e.g. $A \rightarrow XYZ \ \{ \alpha \}$

In a bottom-up parser, the semantic actions α is taken when XYZ is reduced to A. In a top-down parser the action α is taken when $A, X, Y,$ or Z is expanded, whichever is appropriate.

Semantic Action

- In addition to those stated before, the semantic action may also involve:

1. the computation of values for variables belonging to the compiler.
2. the generation of intermediate code.
3. the printing of an error diagnostic.
4. the placement of some values in the symbol table.

Consider the following basic programming-language constructs for generating intermediate codes:

1. Declarations (V)
2. arithmetic assignment operations (V)
3. Boolean expressions (V)
4. flow-of-control statements` if-statement(V) while (V)
5. array references (Δ)
6. procedure calls (V)
7. switch statements (Δ)
8. structure-type references (Δ)

Bottom-up Translation of S-attributed Grammars

- A bottom-up parser uses a stack to hold information about subtrees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes.
- e.g. $A \to XYZ \ \{A.a = f(X.x, Y.y, Z.z)\}$
- Before reduction: the value of the attribute $Z.z$ is in $val\ [top]$, $Y.y$ is in $val\ [top-1]$, and $X.x$ is in $val\ [top-2]$.
- After reduction: top is decremented by 2, $A.a$ is put in $val\ [top]$

For Special Conditions : Hook

- $stmt \rightarrow IF\ cond\ THEN\ stmt\ ELSE\ stmt$

==>

$stmt \rightarrow IF\ cond\ THEN$ { action to emit appropriate conditional jump }

$stmt\ ELSE$ { action to emit appropriate unconditional jump } $stmt$

or

$hook1 \rightarrow \lambda$ { action to emit appropriate conditional jump }

$hook2 \rightarrow \lambda$ { action to emit appropriate unconditional jump }

$stmt \rightarrow IF\ cond\ THEN\ hook1\ stmt\ ELSE\ hook2\ stmt$

Semantic Actions for different language constructs

- Declarations

  e.g. int x, y, z;

  float w, z, s;

Suggested grammar:
(Note: This is a very simple grammar mainly used for explanation.)

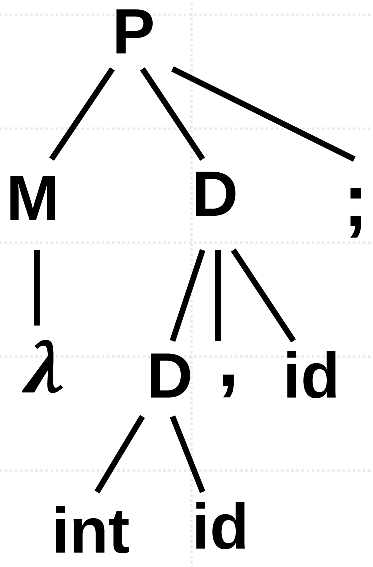
$P \rightarrow MD;$

$M \rightarrow \lambda$ /* empty string */

$D \rightarrow D, id$

$\quad | \quad int\ id$

$\quad | \quad float\ id$



int x , y ;

(Syntax-directed) Translation

$P \rightarrow MD;$ {/* do nothing */}
$M \rightarrow \lambda$ { if offset was not initialized then $offset = 0;$}

$D \rightarrow int\ id$ { **enter** $(\boldsymbol{id.name, int, offset})$**;**
/* a function entering type "int" and
particular offset to the entry $id.name$
of the symbol table */
$D.type = int;$
$offset = offset + 4;$ /*bytes, width of int*/
$D.offset = offset;$ }

$D \rightarrow float\ id$ { enter ($id.name, float, offset$);

$D.type = float$;

$offset = offset + 8$;

/*bytes, width of float*/

$D.offset = offset$; }

$D \rightarrow D_1, id$ { enter ($id.name, D_1.type, D_1.offset$);

$D.type = D_1.type$;

If $D_1.type == int$

$D.offset = D_1.offset + 4$;

else if $D_1.type == float$

$D.offset = D_1.offset + 8$;

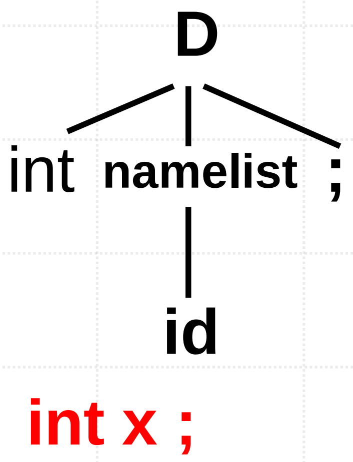
$offset = D.offset$;}

Note: We can construct a data structure to store the information (attributes) of D. (i.e., $D.type$ and $D.offset$)

Avoided grammar:

$D \rightarrow int\ namelist\ ;\ |\ float\ namelist\ ;$

$namelist \rightarrow id,\ namelist\ |\ id$



Why?

When the 'id' is reduced into $namelist$, we cannot know the type of 'id' (int or float?) immediately. Therefore, it is troublesome to enter such type information into the corresponding field of the 'id' in the symbol table. Hence, we must use special coding technique (e.g. linked list keeping the ids name (pointers to symbol table) to achieve such a purpose. (* In other words, we need backpatch to chain the data type.)

Acceptable grammar

$$D \rightarrow \ int\ intlist\ ; \ \ |\, float\ floatlist\ ;$$
$$intlist \rightarrow \ id, intlist \ \ |\, id$$
$$floatlist \rightarrow id, floatlist \ \ |\, id$$

Advantage: The above-mentioned problem will not happen. That is, when 'id' is reduced, we can identify the type of id. (If id is reduced to $intlist$, then id is of "int" type)

Defect: too much production will occur. => too many states => bad performance

Grammar with cloned productions

$$
\begin{array}{llll}
1 & \text{Start} & \rightarrow & \text{Num}_{ans}\ \$ \\
 & & & \quad \textbf{call}\ \text{PRINT}(ans) \\
2 & \text{Num}_{ans} & \rightarrow & \text{o}\ \ \text{OctDigs}_{octans} \\
 & & & \quad ans \leftarrow octans \\
3 & & | & \text{DecDigs}_{decans} \\
 & & & \quad ans \leftarrow decans \\
4 & \text{DecDigs}_{up} & \rightarrow & \text{DecDigs}_{below}\ \ \text{d}_{next} \\
 & & & \quad up \leftarrow below \times 10 + next \\
5 & & | & \text{d}_{first} \\
 & & & \quad up \leftarrow first \\
6 & \text{OctDigs}_{up} & \rightarrow & \text{OctDigs}_{below}\ \ \text{d}_{next} \\
 & & & \quad \textbf{if}\ next \geq 8 \\
 & & & \quad \textbf{then}\ \text{ERROR}(\text{"Non-octal digit"}) \\
 & & & \quad up \leftarrow below \times 8 + next \\
7 & & | & \text{d}_{first} \\
 & & & \quad \textbf{if}\ first \geq 8 \\
 & & & \quad \textbf{then}\ \text{ERROR}(\text{"Non-octal digit"}) \\
 & & & \quad up \leftarrow first
\end{array}
$$

How to handle the following declaration?

$$x, y, z : \ float$$

- Two approaches:

(I) $decl \ \rightarrow \ id\_list \ ':' \ type$

$id\_list \ \rightarrow \ id\_list \ ',' \ id \mid id$

$type \ \rightarrow \ int \mid float$

(II) $decl \rightarrow id \ ':' \ type \mid id \ , decl$

$type \rightarrow int \mid float$

- Which one is better for LR parsing? Why?

Grammar Rule	Semantic Rules
$decl \rightarrow id\_list \, ':' \, type$	$id\_list.dtype = type.dtype$
$id\_list_1 \rightarrow id\_list_2 \, ',' \, id$	$id\_list_2.dtype = id\_list_1.dtype$ $id.dtype = id\_list_1.dtype$
$\mid \; id$	$id.dtype = id\_list.dtype$
$type \rightarrow int$	$type.dtype = int$
$\mid float$	$type.dtype = float$

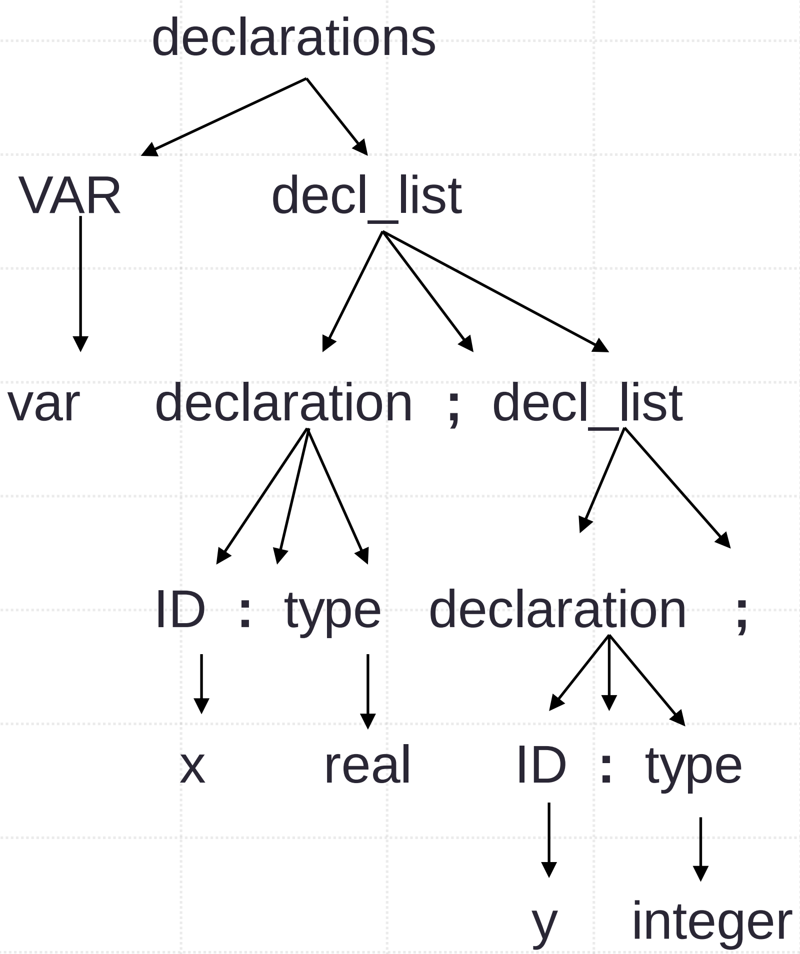
Grammar Rule	Semantic Rules
$decl \rightarrow id\ ':'\ type$	$id.dtype = type.dtype$ $decl.dtype = type.dtype$
$\quad \mid id\ ,decl$	$id.dtype = decl.dtype$
$type \rightarrow int$	$type.dtype = int$
$\quad \mid float$	$type.dtype = float$

Suggested Grammar for the following Declaration:

var x,y,z : real; u,v,t : integer; …

```
declarations :  VAR  decl_list
             |  /*  empty (no declaration is permitted) */
             ;
decl_list    : declaration ';'
             | declaration ';' decl_list
             ;
declaration  : ID ':' type
             | ID ',' declaration
             ;
type         : REAL
             | INTEGER
             ;
```

Try to construct a parse tree for the following declaration and see how to parse it: var x: real; y: integer;



The following grammar for declaration is difficult for attribute gathering.

```
declaration   : id_list ':' type ;
id_list       : ID
              | id_list ',' ID
type          : REAL
              | INTEGER
```

e.g.,