CS 4300: Compiler Theory

Chapter 5
Syntax-Directed Translation

Xuejun Liang

2019 Fall

Outlines (Sections)

1. Syntax-Directed Definitions
2. Evaluation Orders for SDD's
3. Applications of Syntax-Directed Definition
4. Syntax-Directed Translation Schemes
5. Implementing L-Attributed SDD's

1. Syntax-directed Definition

- A syntax-directed definition (SSD) specifies the values of attributes by associating semantic rules with the grammar productions

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel {}'+'$

- A syntax-directed translation scheme embeds program fragments called semantic actions within production bodies

$$E \rightarrow E_1 + T \quad \{ \text{print } '+' \}$$

- Between the two notations
  - syntax-directed definitions can be more readable, and hence more useful for specifications.
  - However, translation schemes can be more efficient, and hence more useful for implementations

Attributes

- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- An inherited attribute at node N is defined only in terms of attribute values at N's parent , N itself, and N's siblings
- Attribute values typically represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Example Syntax-directed Definition

A simple desk calculator

Production	Semantic Rule
$L \rightarrow E$ **n**	$L.val = E.val$
$E \rightarrow E_1 + T$	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
$F \rightarrow$ **digit**	F.val = **digit**.lexval

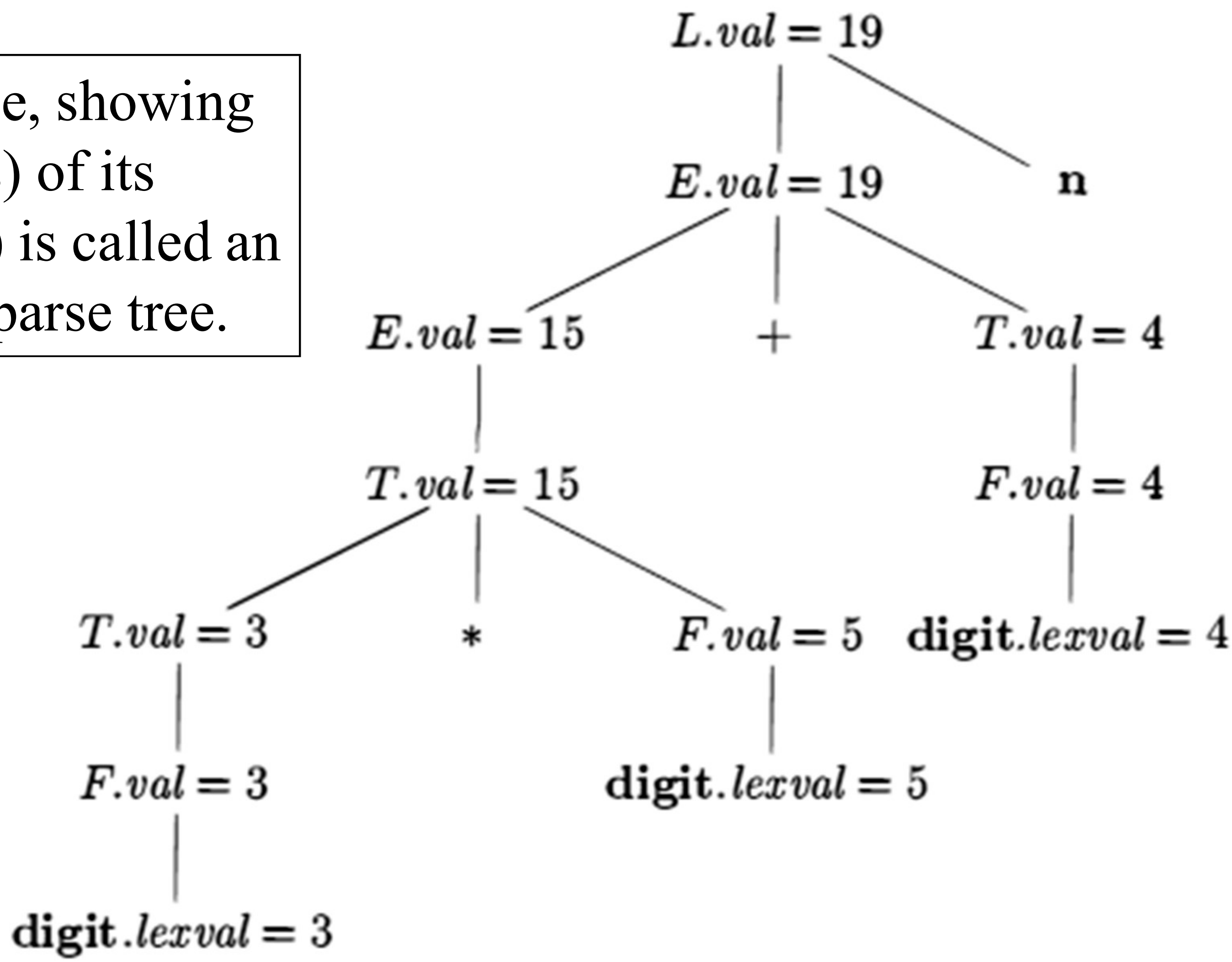
An SDD with only synthesized attributes is called S-attributed.

An SDD without side effects is called an attribute grammar

Note: all attributes in this example are of the synthesized type

Annotated Parse Tree for 3 * 5 + 4 **n**

A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.



Annotating a Parse Tree With Depth-First Traversals

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

procedure *visit*(*n* : *node*);
begin
    for each child *m* of *n*, from left to right **do**
        visit(*m*);
    evaluate semantic rules at node *n*
end

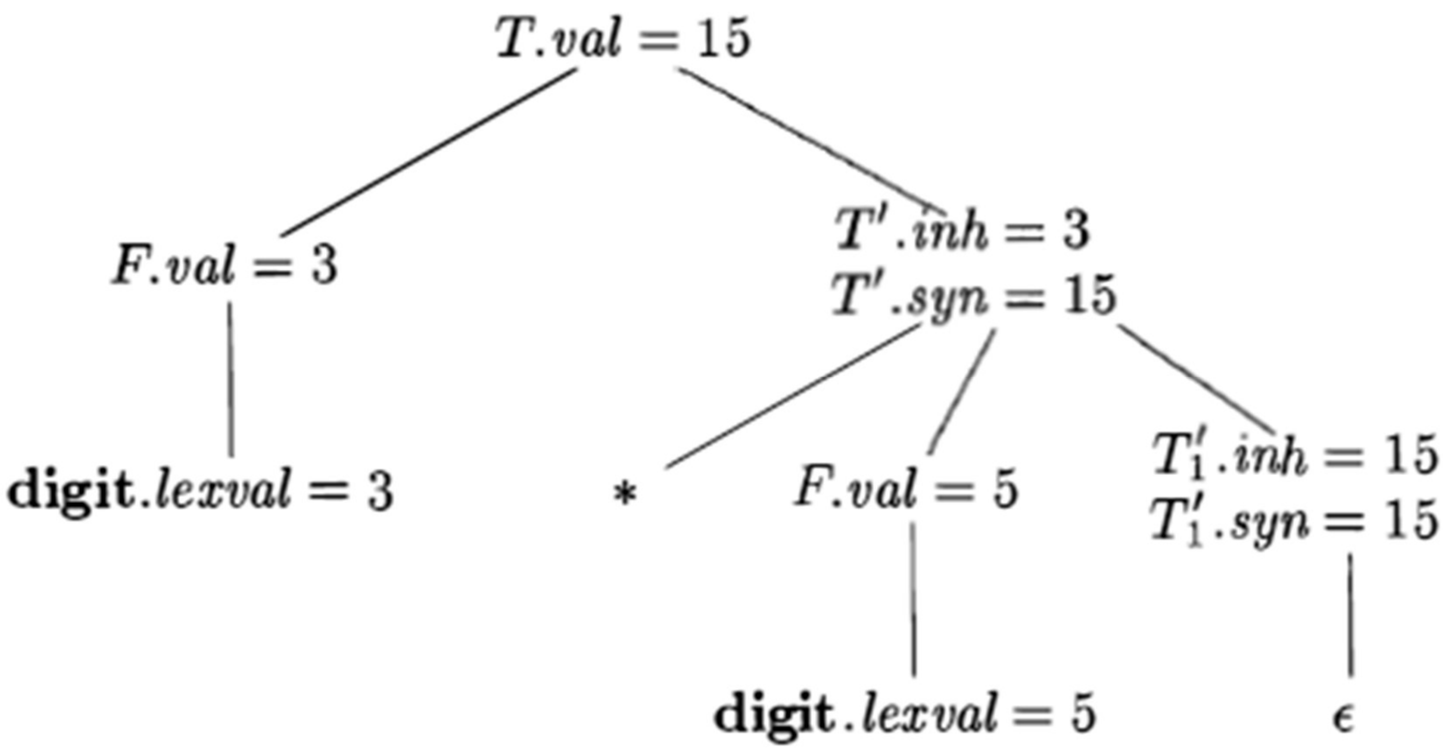
An SDD Based on a Grammar Suitable for Top-down Parsing

	Production	Semantic Rules
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow \mathbf{digit}$

An inherited attribute for nonterminal T' is used to pass the operand to the operator

Annotated Parse Tree for 3 * 5



An inherited attribute for nonterminal T' is used to pass the operand to the operator

Example Attribute Grammar with Synthesized & Inherited Attributes

Simple Type Declaration

Production	Semantic Rule
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ **int**	T.type = '*integer*'
$T \rightarrow$ **float**	T.type = '*float*'
$L \rightarrow L_1$ **, id**	L_1.inh = L.inh; *addtype*(**id**.entry, L.inh)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.inh)

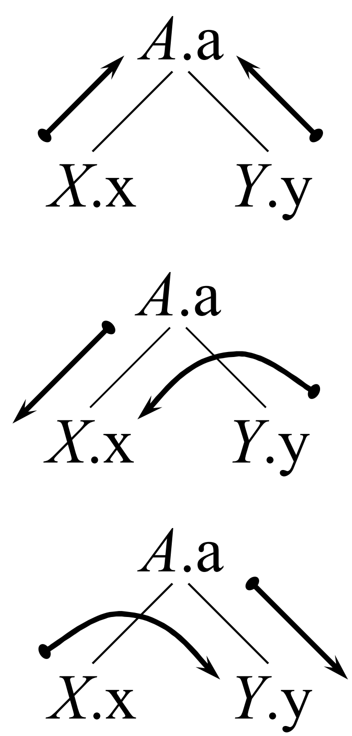
treated as dummy synthesized attribute

Synthesized: T.type, **id**.entry
Inherited: L.inh

2. Evaluation Orders for SDD 's

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree

$A \rightarrow X\,Y$



$A.\mathrm{a} = f(X.\mathrm{x}, Y.\mathrm{y})$

$X.\mathrm{x} = f(A.\mathrm{a}, Y.\mathrm{y})$

$Y.\mathrm{y} = f(A.\mathrm{a}, X.\mathrm{x})$

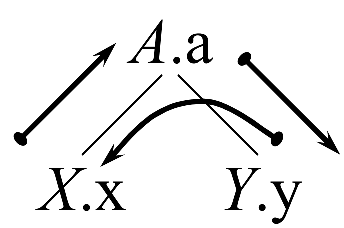
Direction of

value dependence

Evaluation Orders for SDD 's (Cont.)

- Edges in the dependency graph determine the evaluation order for attribute values
  - Dependency graphs cannot be cyclic
- So, dependency graph is a directed acyclic graph (DAG)

$A \to X\ Y$



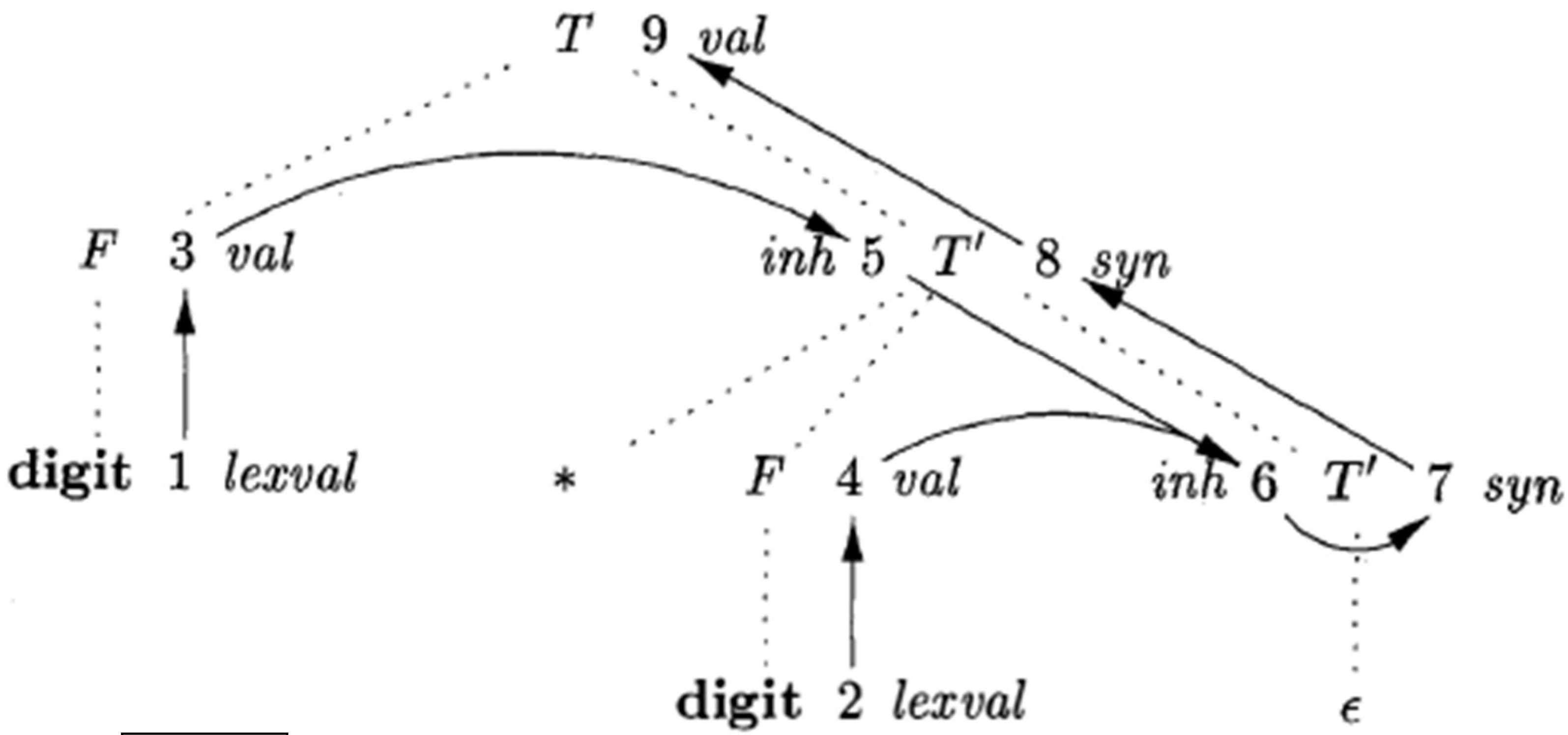
$A.\mathrm{a} := f(X.\mathrm{x})$
$X.\mathrm{x} := f(Y.\mathrm{y})$
$Y.\mathrm{y} := f(A.\mathrm{a})$

Direction of

value dependence

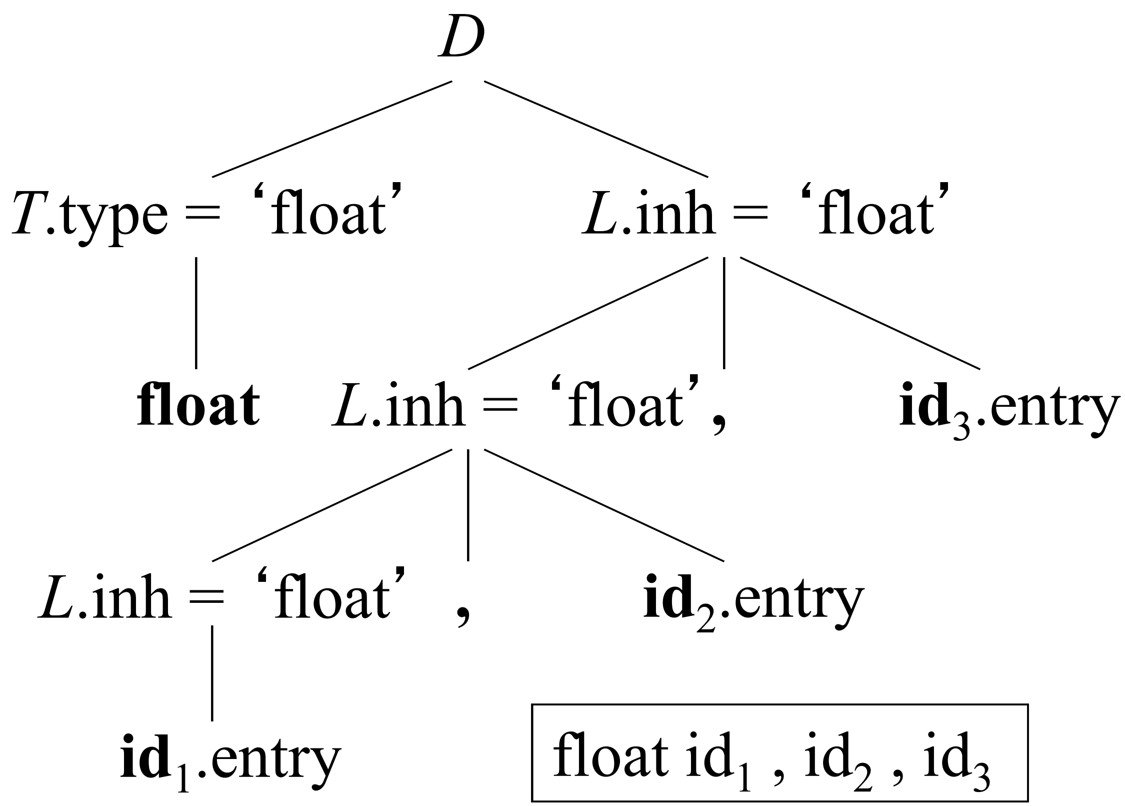
Error: cyclic dependence

Example Annotated Parse Tree with Dependency Graph

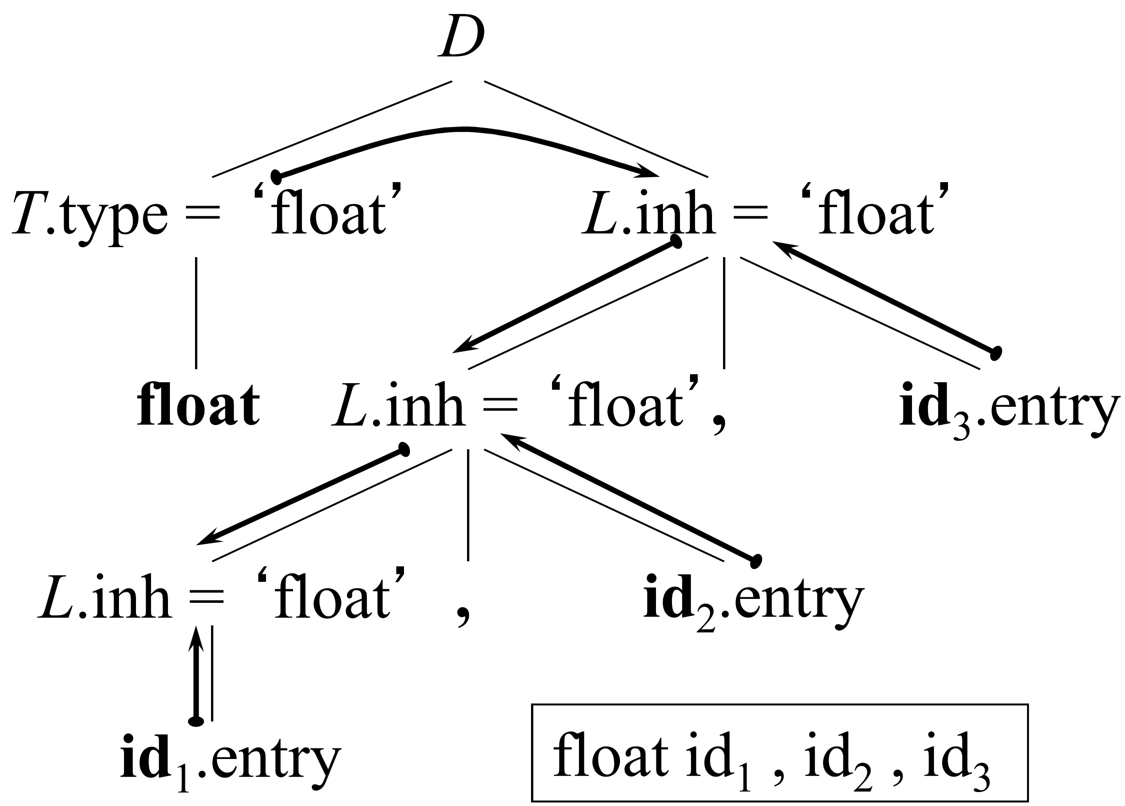


3 * 5

Example Annotated Parse Tree



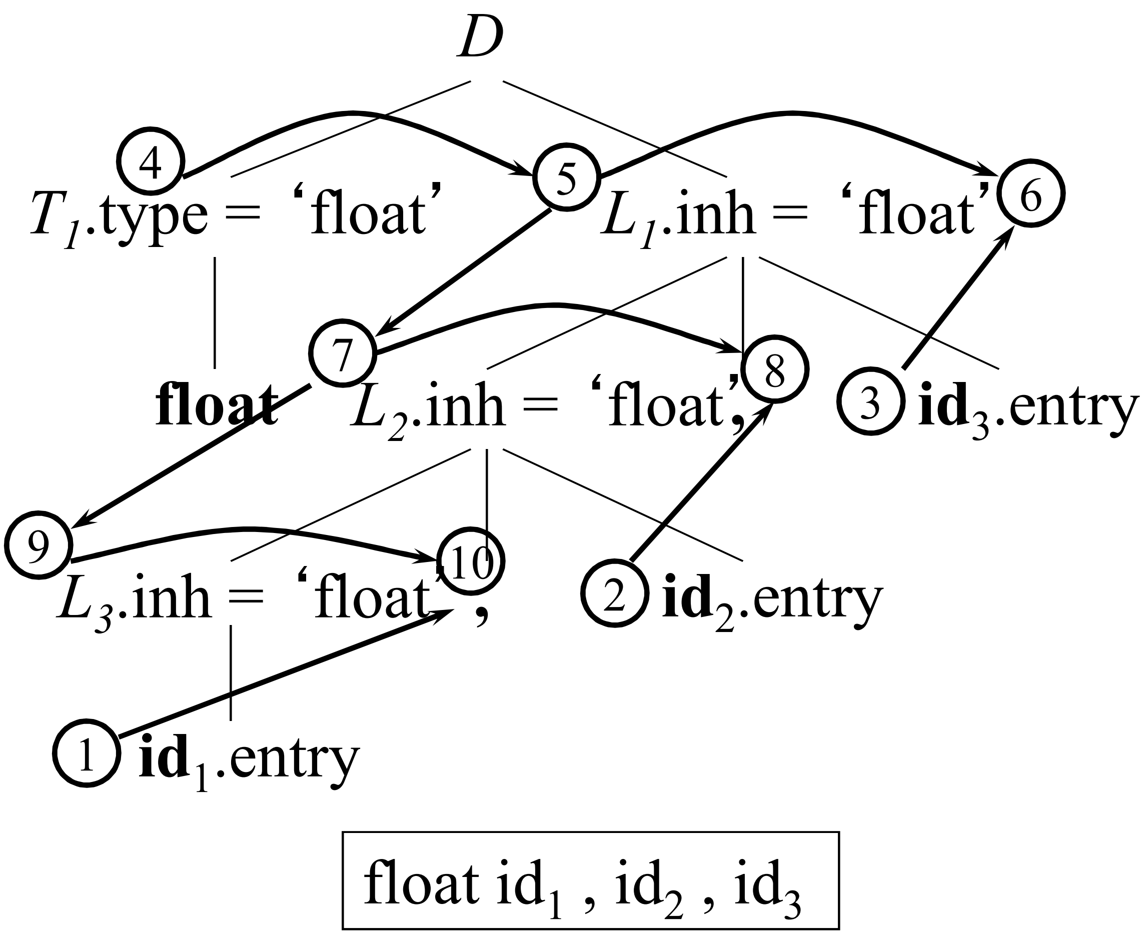
Example Annotated Parse Tree with Dependency Graph



Evaluation Order

- A **topological sort** of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge, then m_i appears before m_j
- Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules
- Example: Topological orders of DAG on slide 13
  - 1, 2, 3, 4, 5, 6, 7, 8, 9.
  - 1, 3, 5, 2, 4, 6, 7, 8, 9.

Example Parse Tree with Topologically Sorted Actions



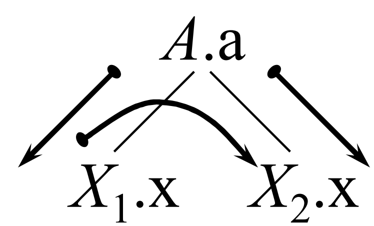
Topological sort:

1. Get $\mathbf{id}_1$.entry
2. Get $\mathbf{id}_2$.entry
3. Get $\mathbf{id}_3$.entry
4. T_1.type= 'float'
5. L_1.inh=T_1.type
6. *addtype*($\mathbf{id}_3$.entry, L_1.inh)
7. L_2.inh=L_1.inh
8. *addtype*($\mathbf{id}_2$.entry, L_2.inh)
9. L_3.inh=L_2.inh
10. *addtype*($\mathbf{id}_1$.entry, L_3.inh)

L-Attributed Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j on the right side of production $A \rightarrow X_1\ X_2\ \ldots\ X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

Shown: dependences of inherited attributes



- L-attributed definitions allow for a natural order of evaluating attributes: **depth-first and left to right**
- Note: every S-attributed syntax-directed definition is also L-attributed

3. Applications of SDD Construction of Syntax Trees

S-attributed Definition for Simple Expressions

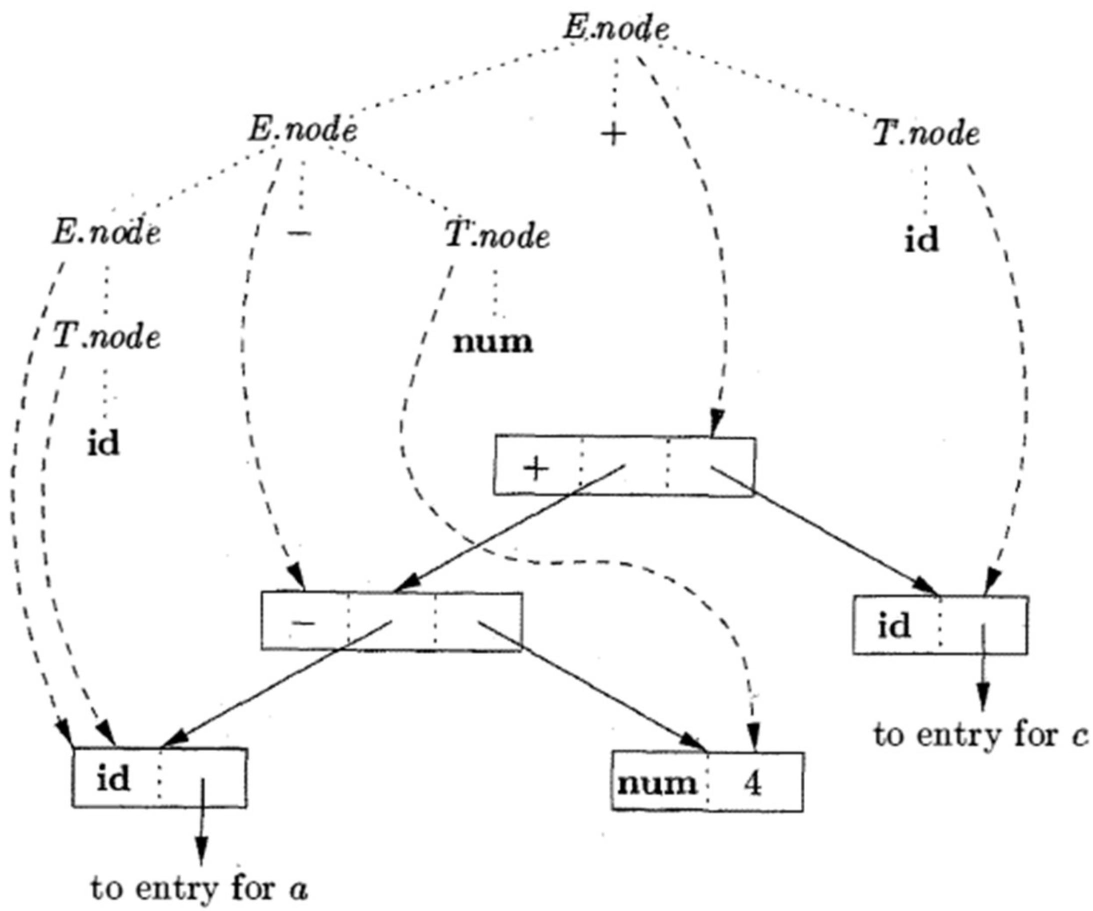
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Note: This is a S-attributed definition, then can be done during bottom-up parsing

Example: Syntax Tree for a - 4 + c

Steps in the construction of the syntax tree

1) $p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

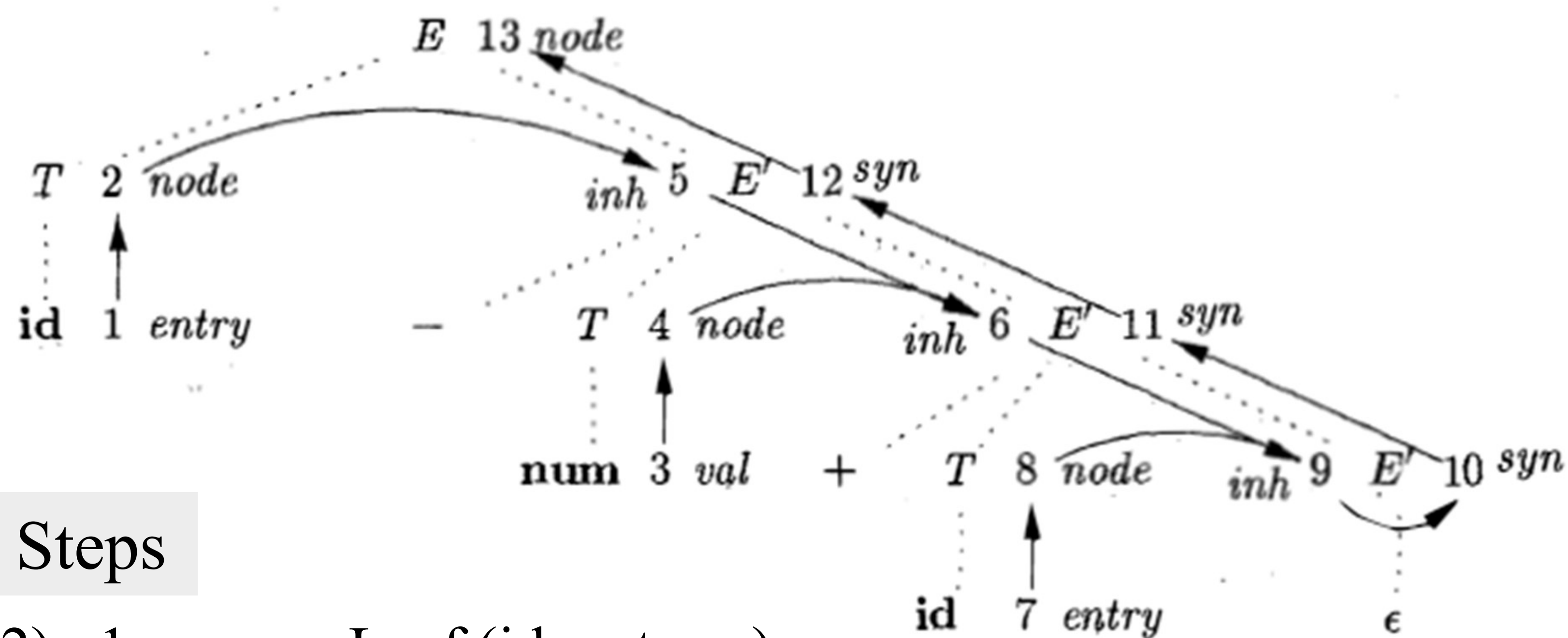


Constructing Syntax Tree During Top-Down Parsing

L-attributed Definition for Simple Expression

	Production	Semantic Rules
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Example: Dependency Graph for a-4+c

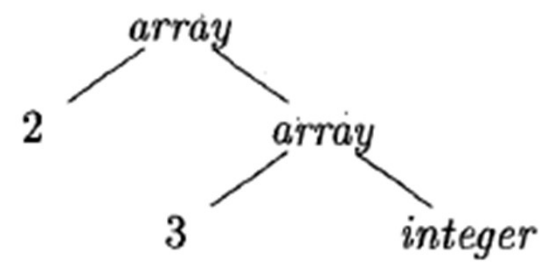


Steps

2) p1 == new Leaf (id, entry-a) ;
4) p2 == new Leaf (num, 4) ;
6) p3 == new Node('-', pl, p2) ;
8) p4 == new Leaf (id, entry-c) ;
9) p5 == new Node('+', p3, p4) ;

The Structure of a Type

int[2][3] → array(2, array(3, integer)) → Type expression for **int[2][3]**



PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

T generates either a basic type or an array type

Annotated Parse Tree for int[2][3]