Syntax-Directed Translation

What is syntax-directed translation?

- The compilation process is driven by the syntax.
- The semantic routines perform interpretation based on the syntax structure.
- Attaching **attributes** to the grammar symbols.
- Values for attributes are computed by **semantic actions** associated with the grammar productions.

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

Format for writing syntax-directed definitions

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

SDD for a desk calculator

- **E.val** is one of the attributes of E.
- **digit.**lexval is the attribute (integer value) returned by the lexical analyzer
- To avoid confusion, recursively defined nonterminals are numbered on the RHS.
- **Semantic actions are performed** when this production is "used".

Each grammar symbol is associated with a set of attributes computed w.r.t. the parsing tree

〈A,N〉 ⟶ a non terminal **A** labelling a node **N** of the parse tree

- **Synthesized attribute of 〈A,N〉** : defined in terms of the attributes of the **children** of **N** and of **N itself** (semantic rule associated to the production relative to **N**)
- **Inherited attribute of 〈A,N〉** : defined in terms of the **N's parent, N itself**, and **N's siblings** (semantic rule associated to the production relative to the parent of **N**)
- **General attribute**: value can be depended on the attributes of any nodes.

Terminal symbols can have synthetised attributed (computed by the lexical analyzer) **but not inherited attributes**.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

SSD for a desk calculator

In this case each non terminal symbol has a **unique synthesized attribute** *val*

S-attributed SDD

involves only synthesized attributes

In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser.

We work with parse trees
***even though a translator needs not actually build a parse tree*.**

A parse tree + the value(s) of its attribute(s):

annotated parse tree

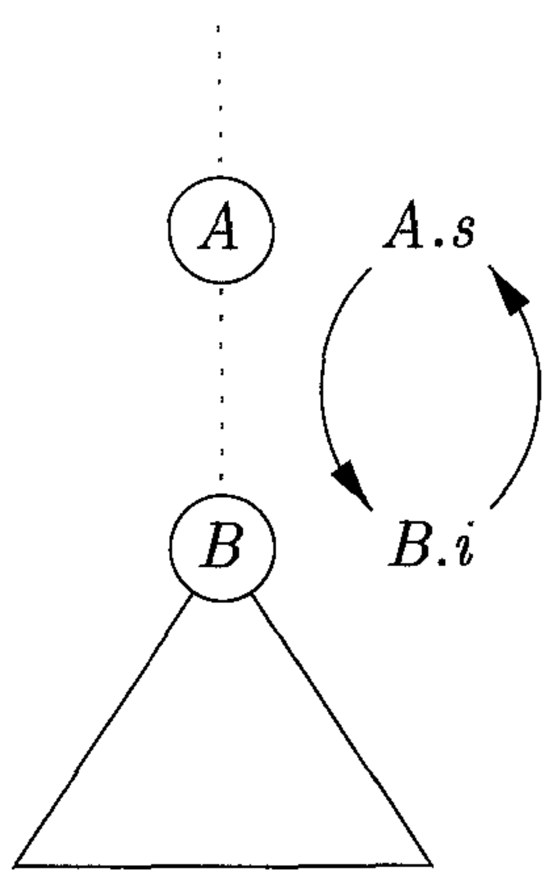
Synthesized attributes: we can evaluate attributes in any bottom-up order, such as a postorder traversal of the parse tree

For SDD's with both inherited and synthesized attributes, there is no guarantee that there exists one order in which to evaluate attributes at nodes

PRODUCTION	*SEMANTIC RULES*
$A \rightarrow B$	$A.s = B.i;\ B.i = A.s + 1$

These rules are circular

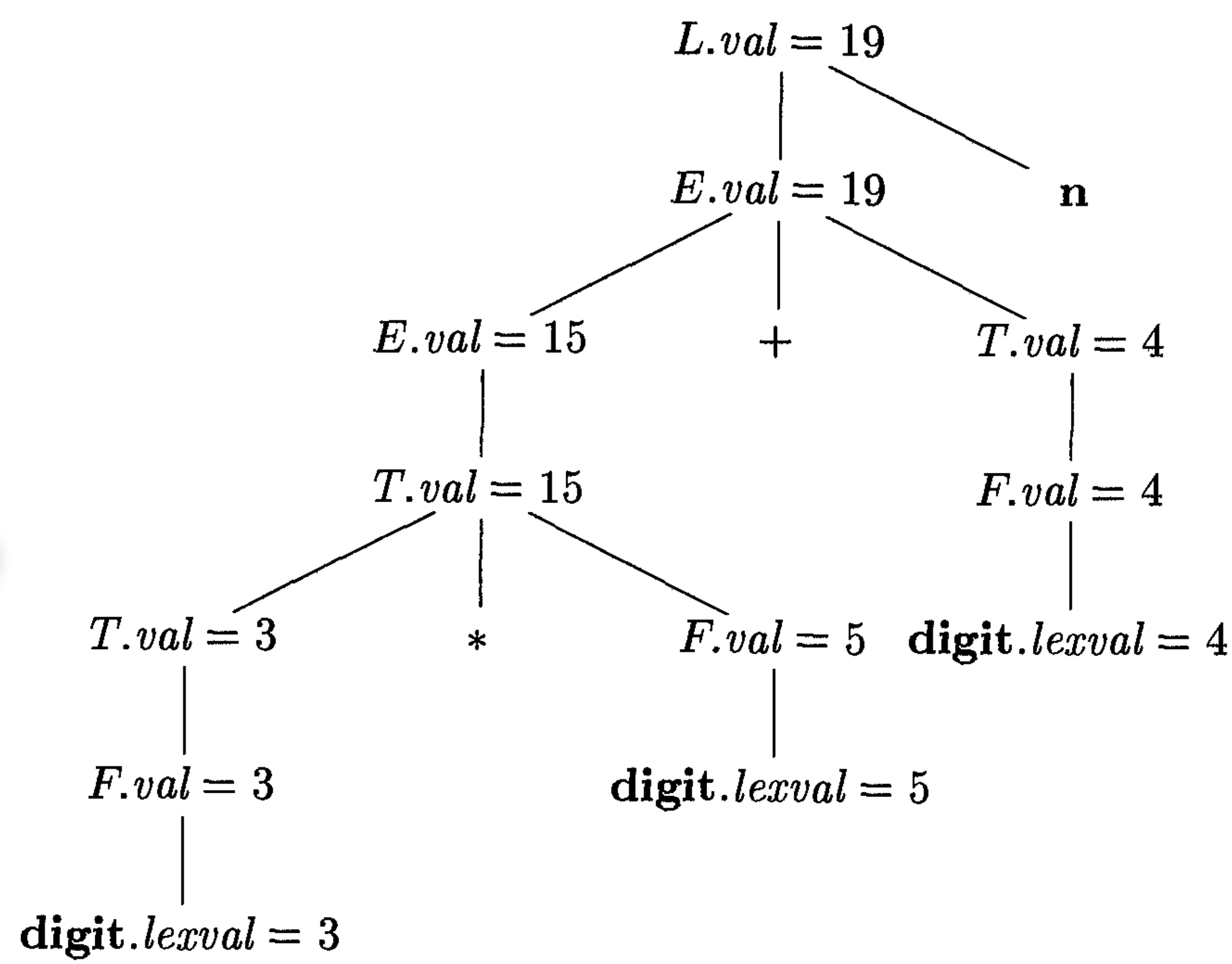
it is impossible to evaluate either A.s at a node N or B.i at the child of N without first evaluating the other



	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

SSD for a desk calculator

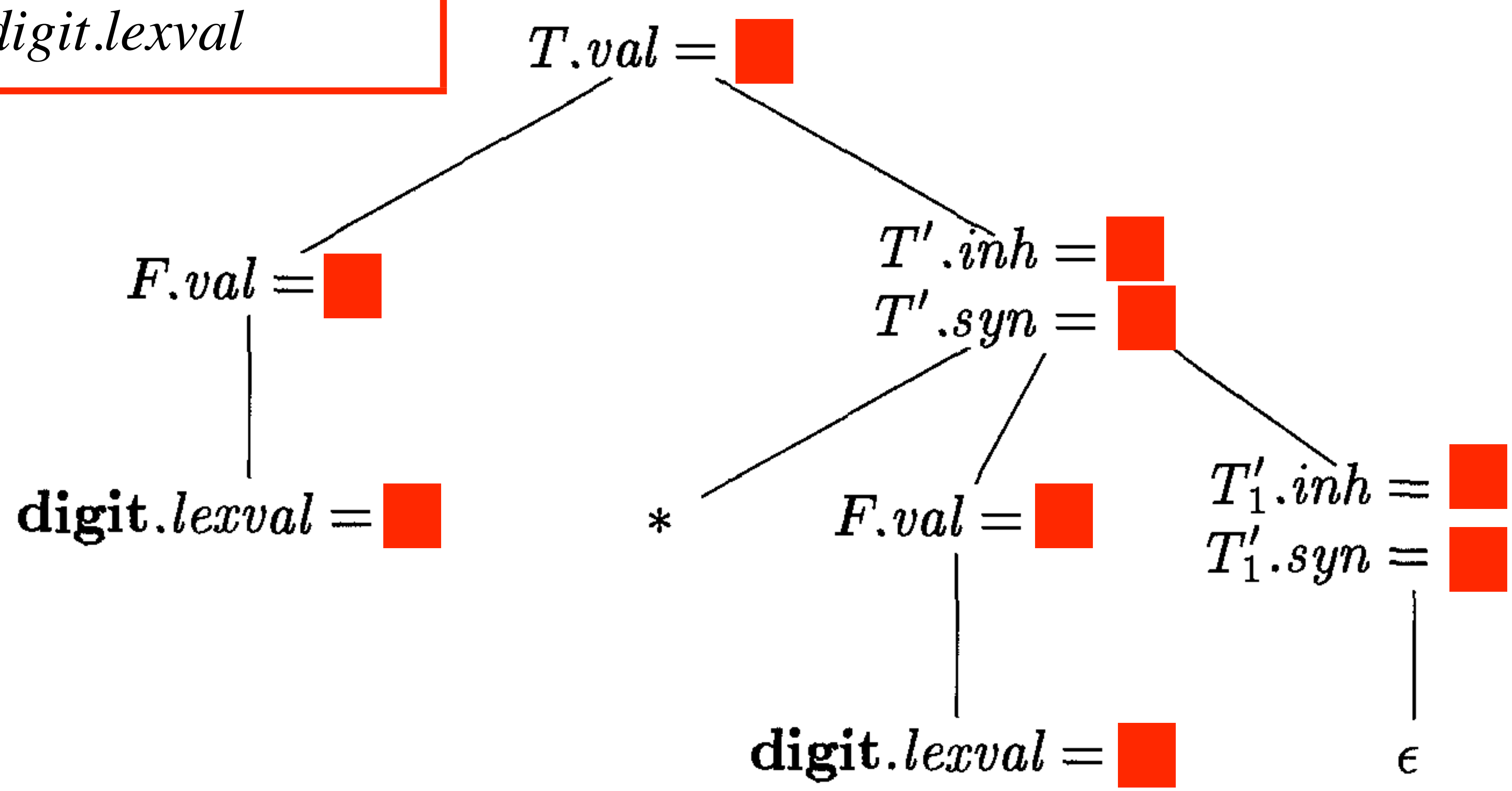
Each of the nodes for the nonterminals has attribute val computed in a bottom-up order



Annotated parse tree for $3 * 5 + 4$ **n**

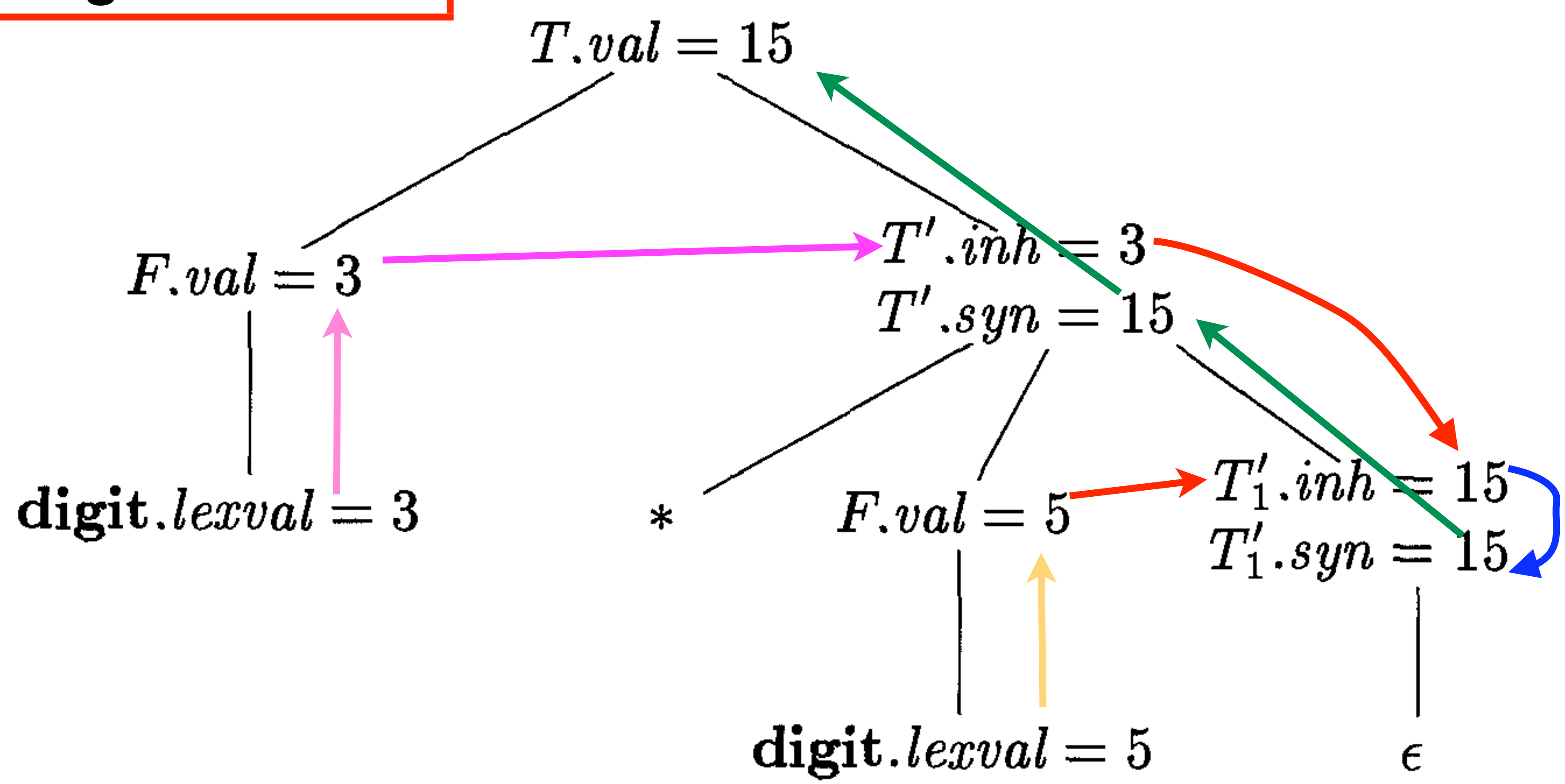
	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow digit$	$F.val = digit.lexval$

The semantic rules are based on the idea that the left operand of the operator * is inherited.
More precisely, the head T' of the production $T' \rightarrow *\ F\ T'_1$ **inherits the left operand of * in the production body**



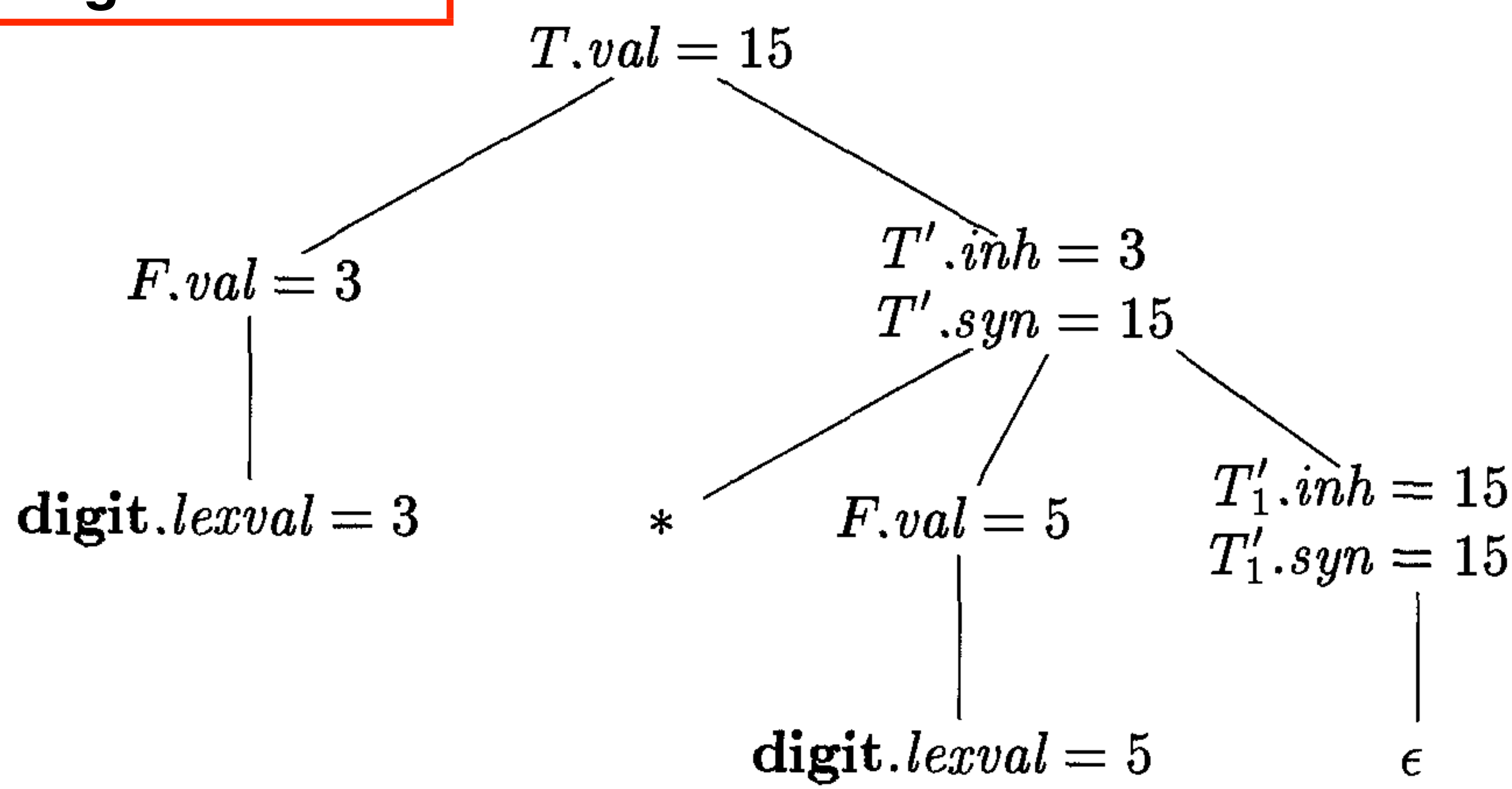
Annotated parse tree for $3 * 5$

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$



Annotated parse tree for $3 * 5$

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$



Annotated parse tree for $3 * 5$

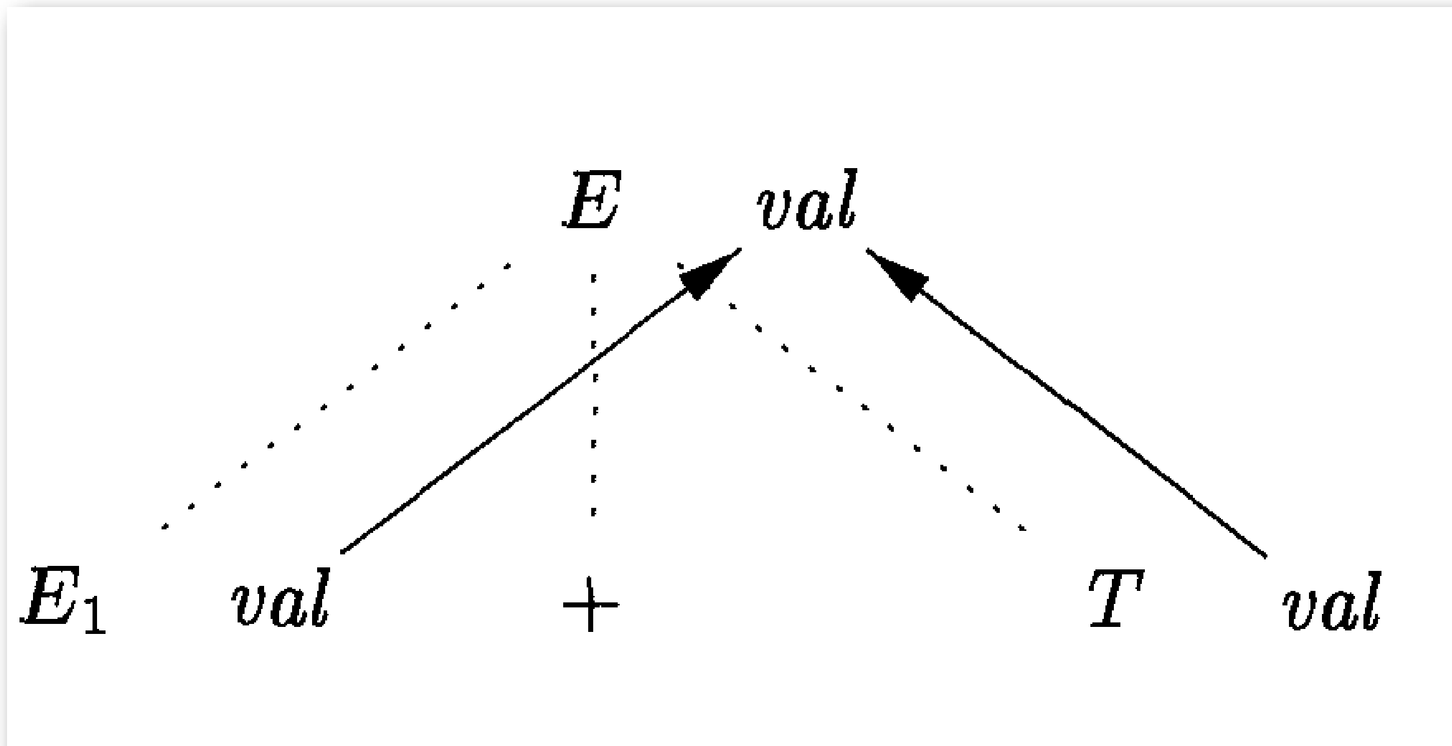
DEPENDENCY GRAPHS

∀ parse-tree-node labeled by X, ∀ X-attribute: the dependency graph has a node.

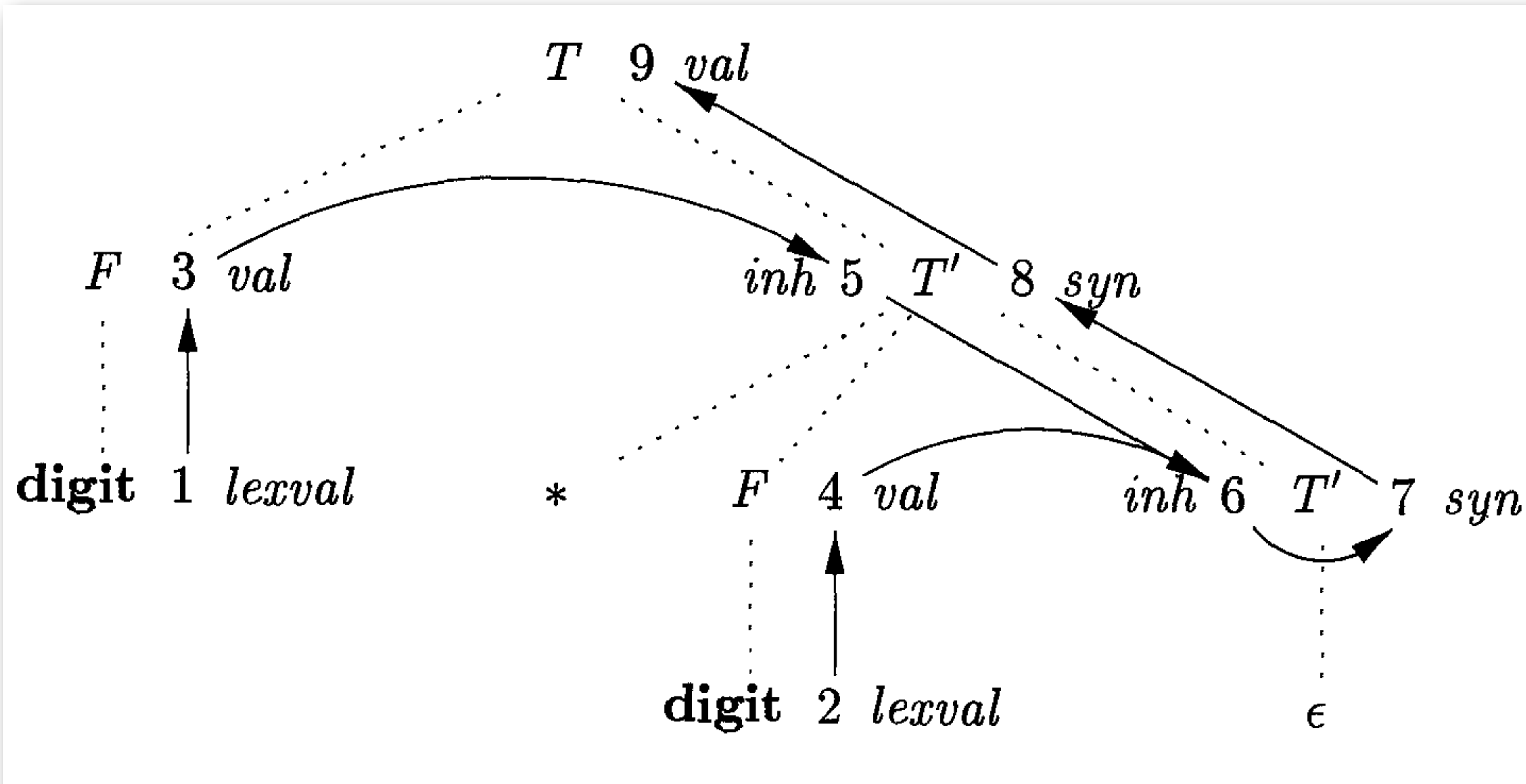
Suppose that a semantic rule associated with a production *p* defines the value of synthesized attribute *A.b* in terms of the value of *X.c* (the rule may define *A.b* in terms of other attributes in addition to *X.c*). Then, the dependency graph has an edge from *X.c* to *A.b.*

Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a.* Then, the dependency graph has an edge from *X.a* to *B.c.*

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$



Ordering the Evaluation of Attributes

If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.

the only allowable orders of evaluation are those sequences of nodes N_1, N_2, ··· .N_i; such that:

if there is an edge of the dependency graph from N_i to N_j, then $i < j$.

Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.

If there are no cycles, however, then there is always at least one topological sort.

S-Attributed Definitions

An SDD is S-attributed if every attribute is synthesized

```
postorder(N){
   foreach (child C of N, from the left)
            postorder(C);
   evaluate the attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing, since a **bottom-up parse corresponds to a postorder traversal**.

Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head.

L-Attributed Definitions

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go **from left to right, but not from right to left** (hence "L-attributed")

Each attribute must be either

1. **Synthesized**

   or

2. **Inherited:**

   if $A \rightarrow X_1 X_2 \ldots X_n$, **and** there is an inherited attribute $X_i.a$ computed by a rule associated with this production **then** the rule may use only:

   (a) **Inherited** attributes associated with the head *A.*

   (b) **inherited** or **synthesized** attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

   (c) **Inherited** or **synthesized** attributes associated with this occurrence of X_i **itself**, but only in such a way that **there are no cycles** in a dependency graph formed by the attributes of this X_i.

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

L-ATTRIBUTED

?

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

L-ATTRIBUTED

Any SDD containing the following production and rules **cannot be L-attributed**:

PRODUCTION	**SEMANTIC RULES**
$A \rightarrow BC$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

SEMANTIC RULES WITH CONTROLLED SIDE EFFECTS

Side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table...

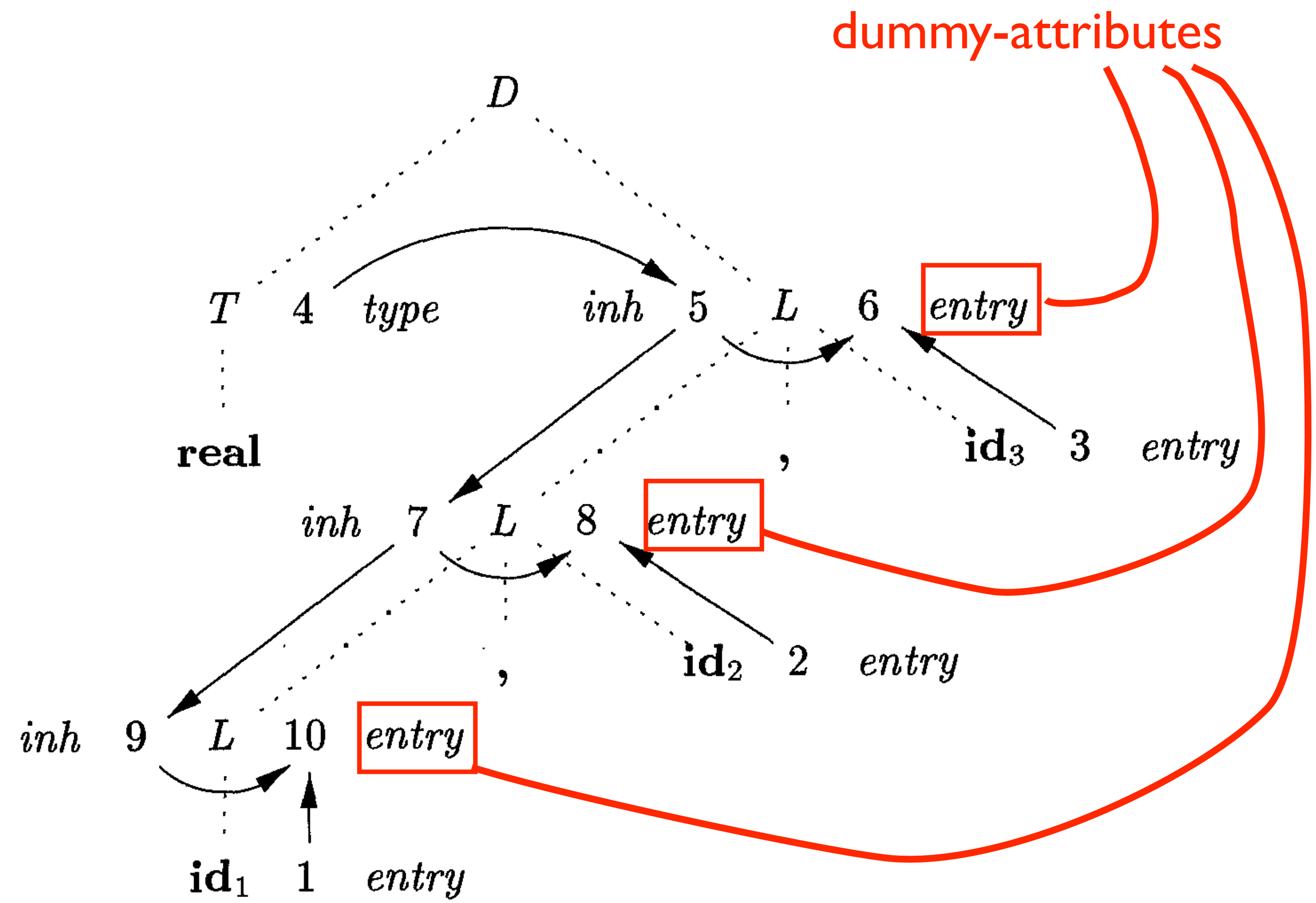
	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$print(E.val)$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

	Production	Semantic Rules
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to \textbf{int}$	$T.type = \text{integer}$
3)	$T \to \textbf{float}$	$T.type = \text{float}$
4)	$L \to L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \to \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. *id.entry,* a lexical value that points to a symbol-table object, and
2. *L. inh,* the type being assigned to every identifier on the list.

We suppose that function *addType* properly installs the type *L.inh* as the type of the represented identifier.

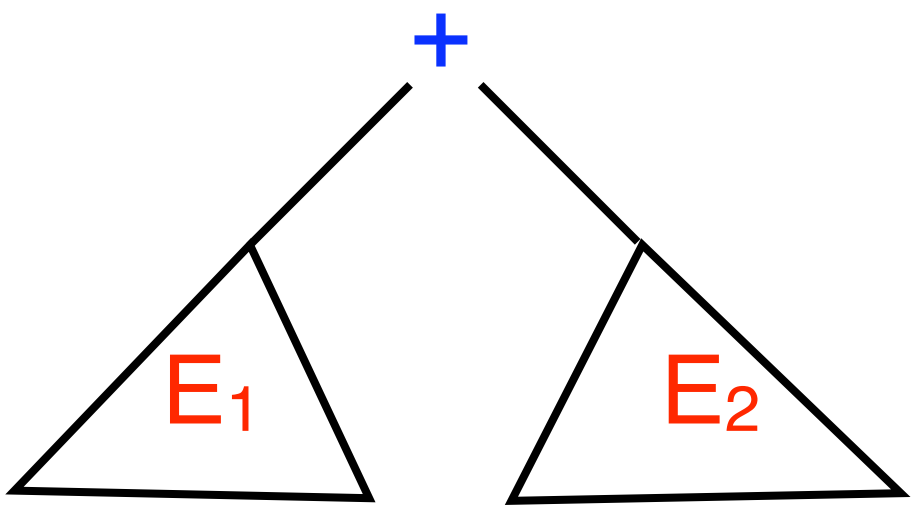


Dependency graph for a declaration float $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$

CONSTRUCTION OF (ABSTRACT) SYNTAX TREES

In an (*abstract*) ***syntax tree*** for an expression, each **interior node** represents an **operator**; the **children** of the node represent the **operands** of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

$E_1 + E_2$

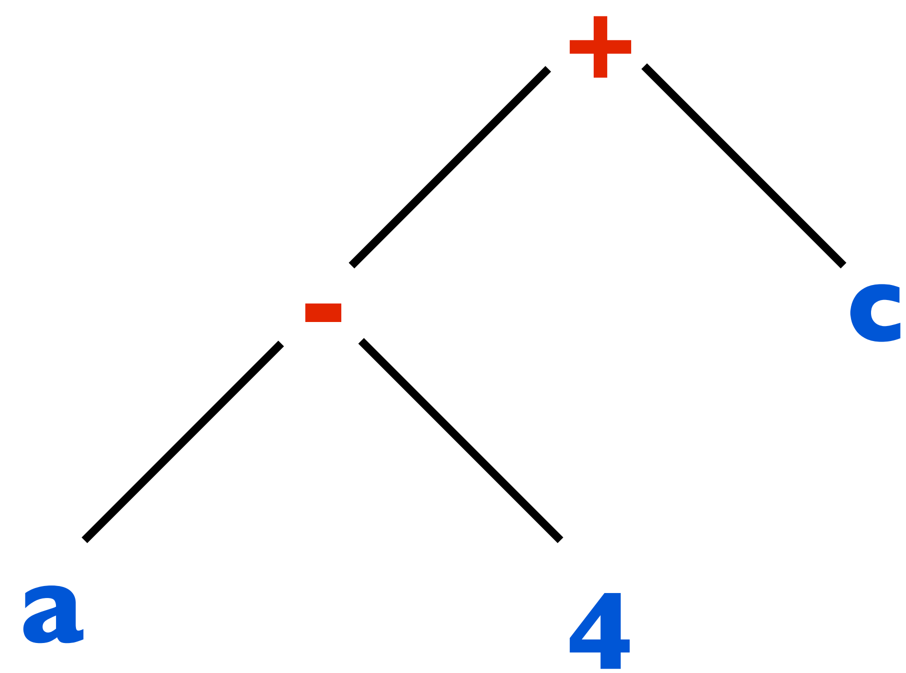


- If the node is a **leaf**, an additional field holds the lexical value for the leaf. A constructor function **Leaf(op, val)** creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.
- If the node is an **interior node**, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: **Node(op, c_1, c_2, ... ,c_k)** creates an object with first field **op** and *k* additional fields for the *k* children **c_1, c_2, ... ,c_k**

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	
2)	$E \to E_1 - T$	
3)	$E \to T$	?
4)	$T \to (E)$	
5)	$T \to \textbf{id}$	
6)	$T \to \textbf{num}$	

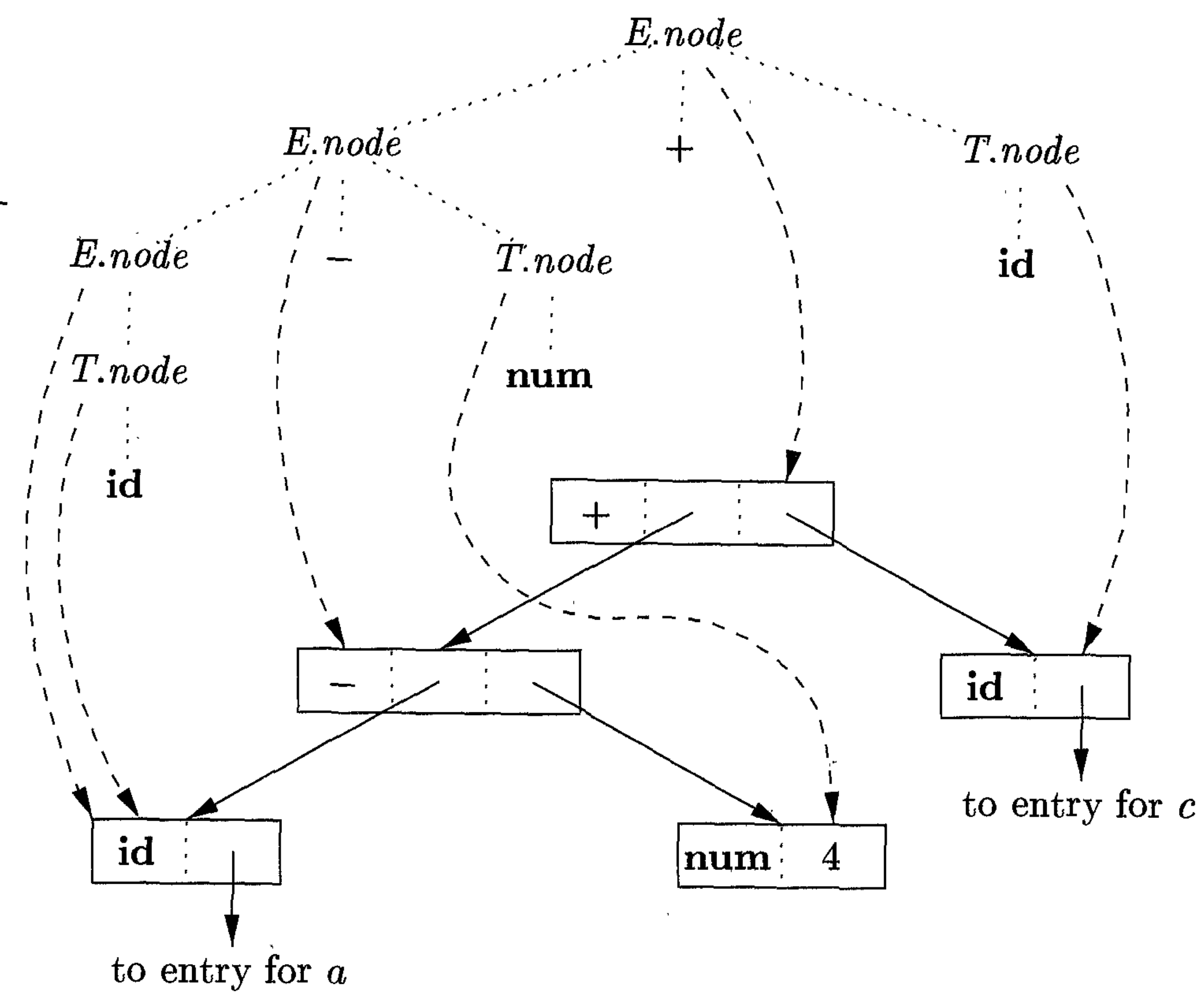
Figure 5.10: Constructing syntax trees for simple expressions

a-4+c



	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)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

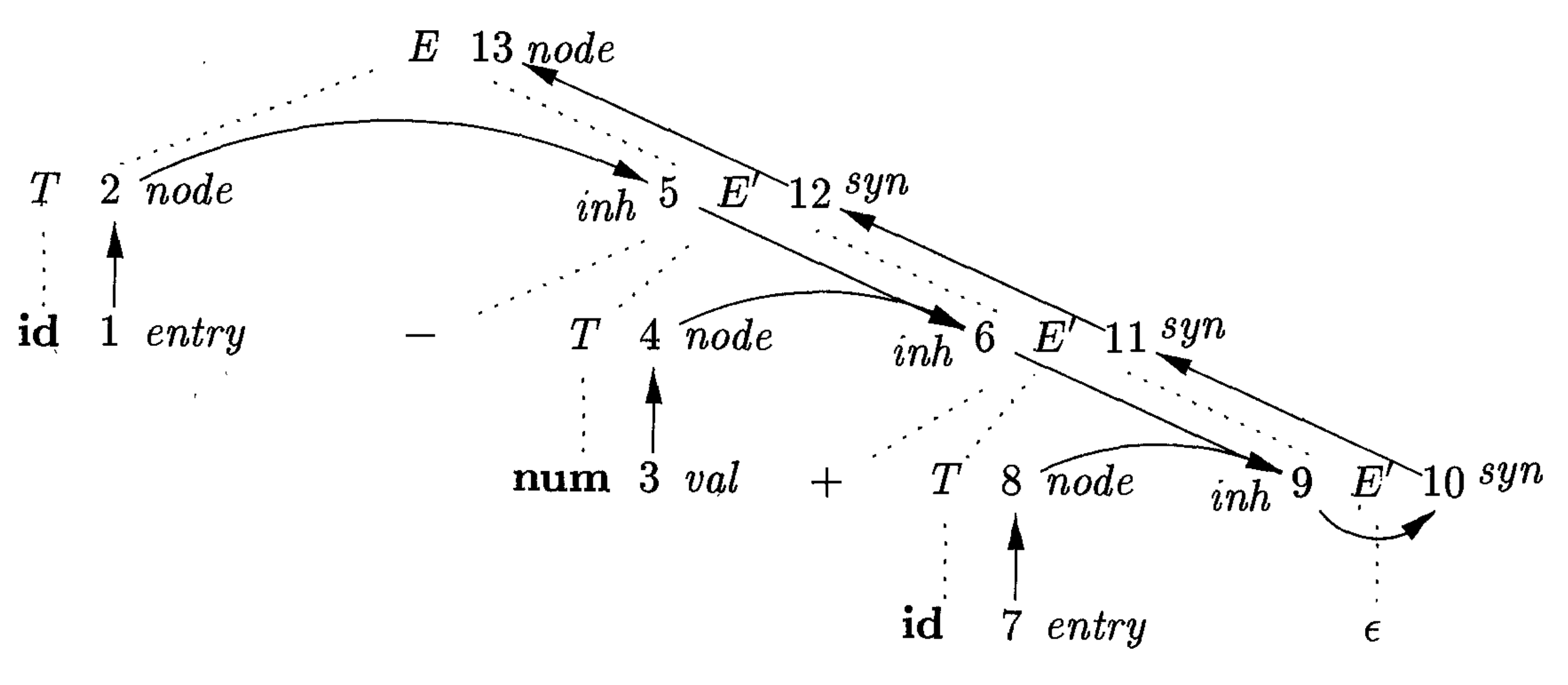


1) $p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

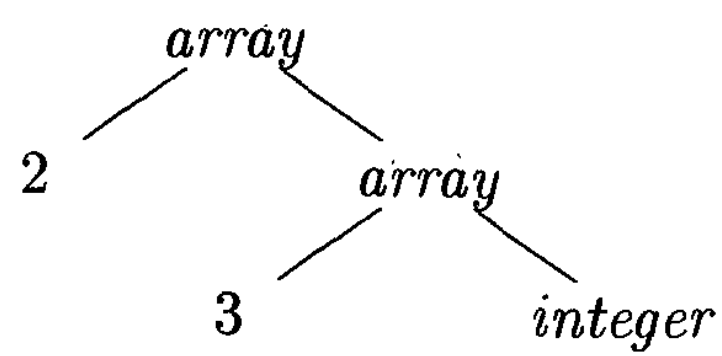
Steps in the construction of the syntax tree for $a - 4 + c$

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

Here, the idea is to build a syntax tree for x + y by passing x as an inherited attribute, since x and + y appear in different subtrees



Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input



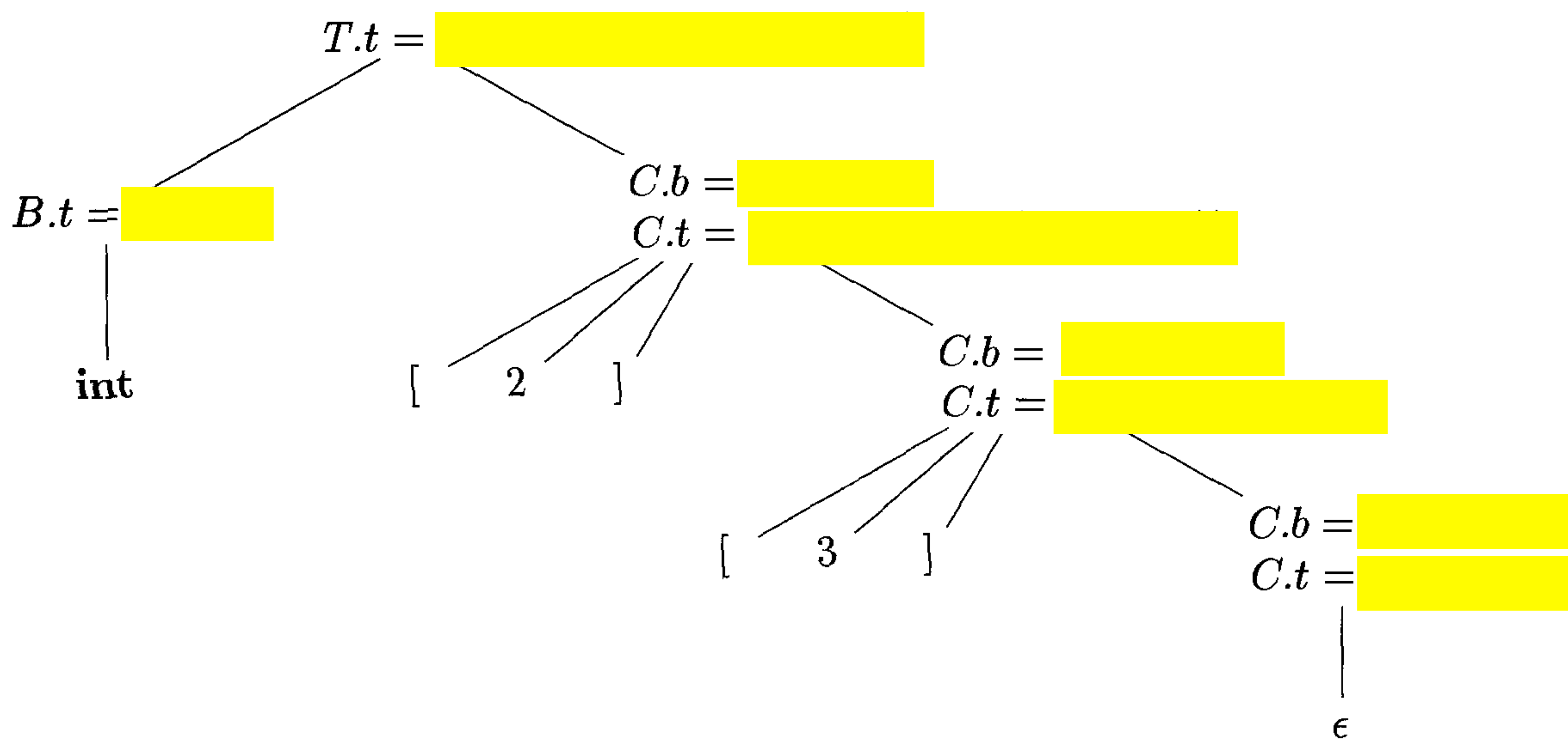
`int[2][3] ≡ array(2,array(3,integer))`

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

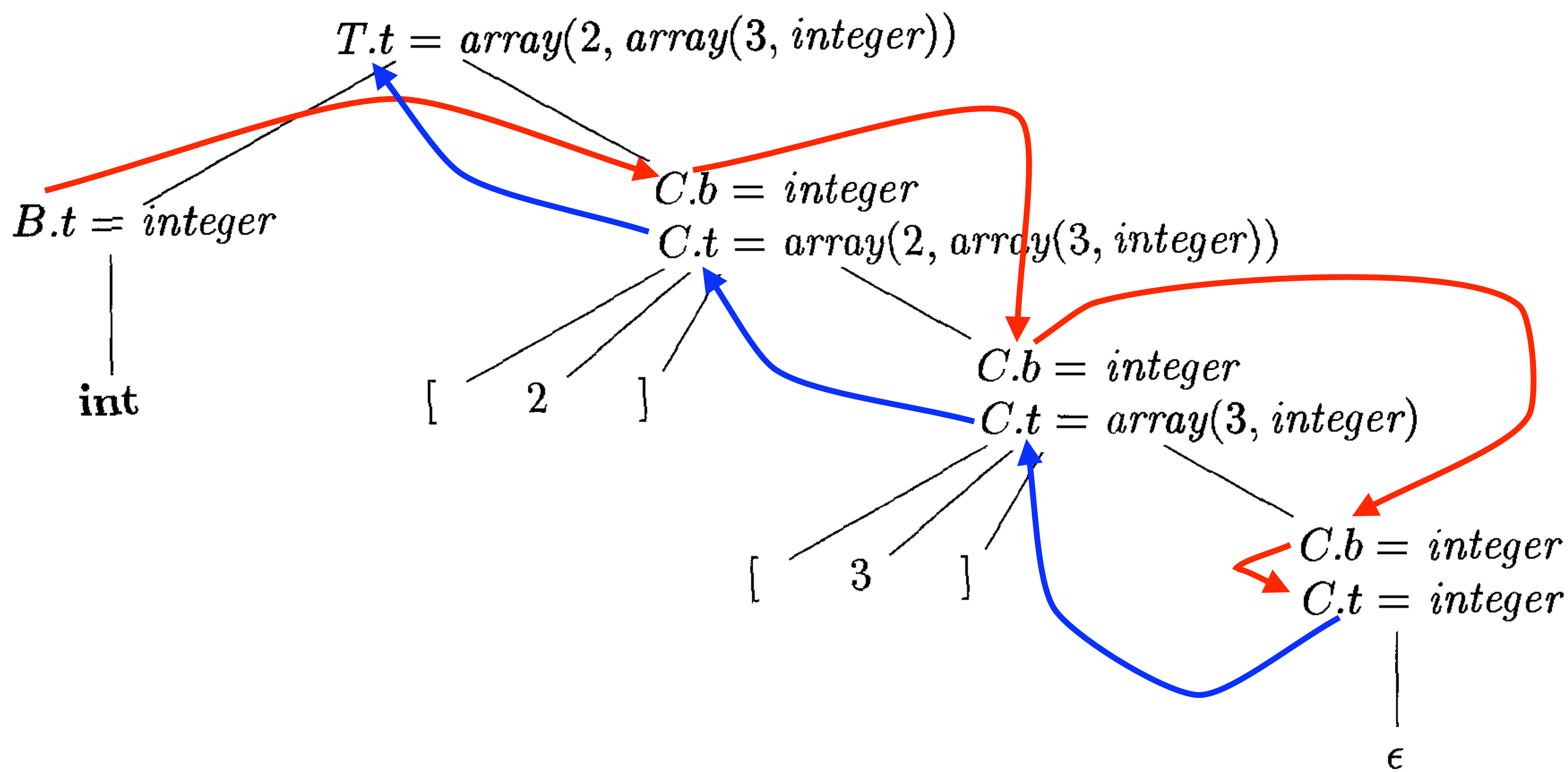
C.b inherited

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

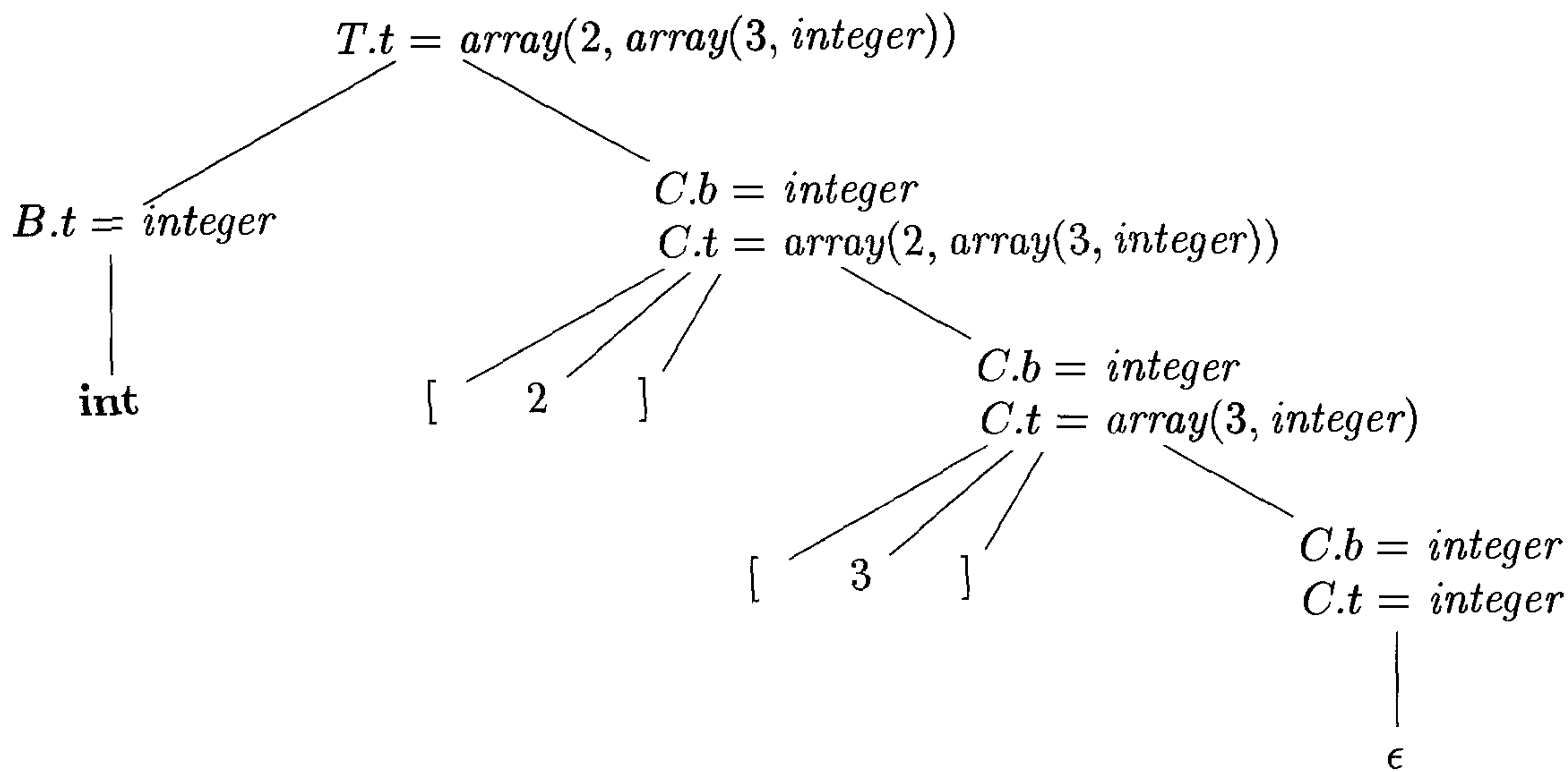
`int[2][3] ≡ array(2,array(3,integer))`



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



Production	Semantic Rules
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



Problems with L-attributed definitions

Comparisons:

- L-attributed definitions go naturally with LL parsers.
- S -attributed definitions go naturally with LR parsers.
- L-attributed definitions are more flexible than S -attributed definitions.
- LR parsers are more powerful than LL parsers.

Some cases of L-attributed definitions cannot be incooperated into LR parsers

- Assume the next handle to take care is $\mathbf{A \rightarrow X_1 X_2 \cdots X_i \cdots X_k}$, and $X_1, \ldots, X_i$ is already on the top of the STACK.
- Attribute values of $X_1, \ldots, X_{i-1}$ can be found on the STACK at this moment.
- No information about A can be found anywhere at this moment.
- Thus the attribute values of X_i cannot be depended on the value of A.

L^--attributed definitions

Same as L-attributed definitions, but do not depend on

- ◁ **the inherited attributes of parent nodes, or**
- ◁ **any attributes associated with itself.**

Can be handled by LR parsers.

Syntax-Directed Translation scheme

Syntax-Directed Translation scheme
(SDT)
=
CFG + program fragments embedded within production bodies

program fragments ➽ **semantic actions**
program fragments can appear at any position within a production body

Typically, SDT's are implemented during parsing, without building a parse tree.

We will see that any SDT can be implemented by:

1) first building a parse tree

and

2) then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

Implementation of two important classes of SDD's by means of SDT

- The underlying grammar is LR-parsable, and the SDD is S-attributed.
- The underlying grammar is LL-parsable, and the SDD is L-attributed.

Postfix Translation Schemes

The simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed.

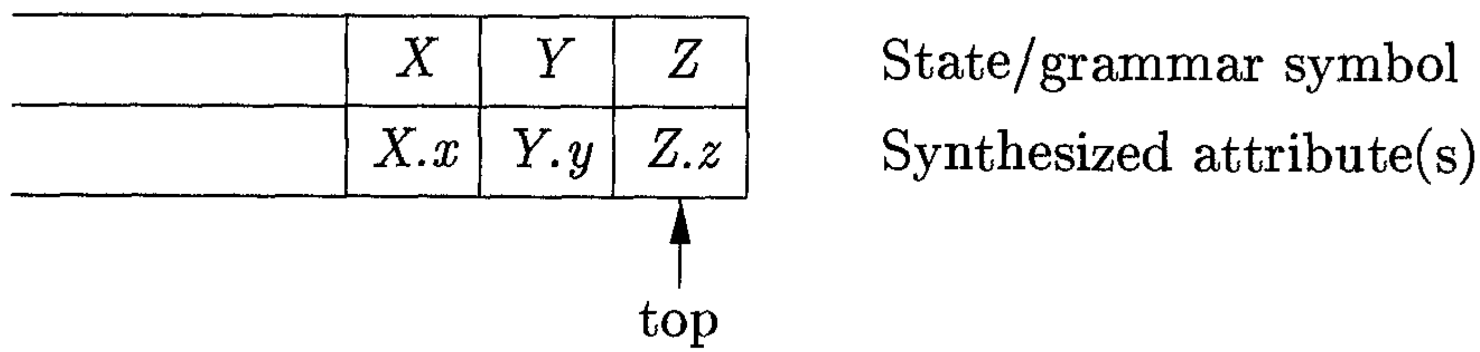
In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

$$
\begin{array}{lcll}
L & \rightarrow & E \ \mathbf{n} & \{ \ \text{print}(E.val); \ \} \\
E & \rightarrow & E_1 + T & \{ \ E.val = E_1.val + T.val; \ \} \\
E & \rightarrow & T & \{ \ E.val = T.val; \ \} \\
T & \rightarrow & T_1 * F & \{ \ T.val = T_1.val \times F.val; \ \} \\
T & \rightarrow & F & \{ \ T.val = F.val; \ \} \\
F & \rightarrow & (\ E\ ) & \{ \ F.val = E.val; \ \} \\
F & \rightarrow & \mathbf{digit} & \{ \ F.val = \mathbf{digit}.lexval; \ \}
\end{array}
$$

Postfix SDT implementing the desk calculator

Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur



Parser stack with a field for synthesized attributes

If the **attributes are all synthesized**, and **the actions occur at the ends of the productions**, then **we can compute the attributes for the head when we reduce the body to the head**.

If we reduce by a production such as $A \rightarrow X\ Y\ Z$, then we have all the attributes of *X, Y,* and *Z* available, at known positions on the stack. After the action, *A* and its attributes are at the top of the stack, in the position of the record for *X.*

PRODUCTION	ACTIONS
$L \to E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \to E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \to T$	
$T \to T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \to F$	
$F \to (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \to \mathbf{digit}$	

Implementing the desk calculator on a bottom-up parsing stack

L	$\to$	$E\ \mathbf{n}$	$\{\ \text{print}(E.val);\ \}$
E	$\to$	$E_1 + T$	$\{\ E.val = E_1.val + T.val;\ \}$
E	$\to$	T	$\{\ E.val = T.val;\ \}$
T	$\to$	$T_1 * F$	$\{\ T.val = T_1.val \times F.val;\ \}$
T	$\to$	F	$\{\ T.val = F.val;\ \}$
F	$\to$	$(\ E\ )$	$\{\ F.val = E.val;\ \}$
F	$\to$	$\mathbf{digit}$	$\{\ F.val = \mathbf{digit}.lexval;\ \}$

An extreme example of a problematic SDT

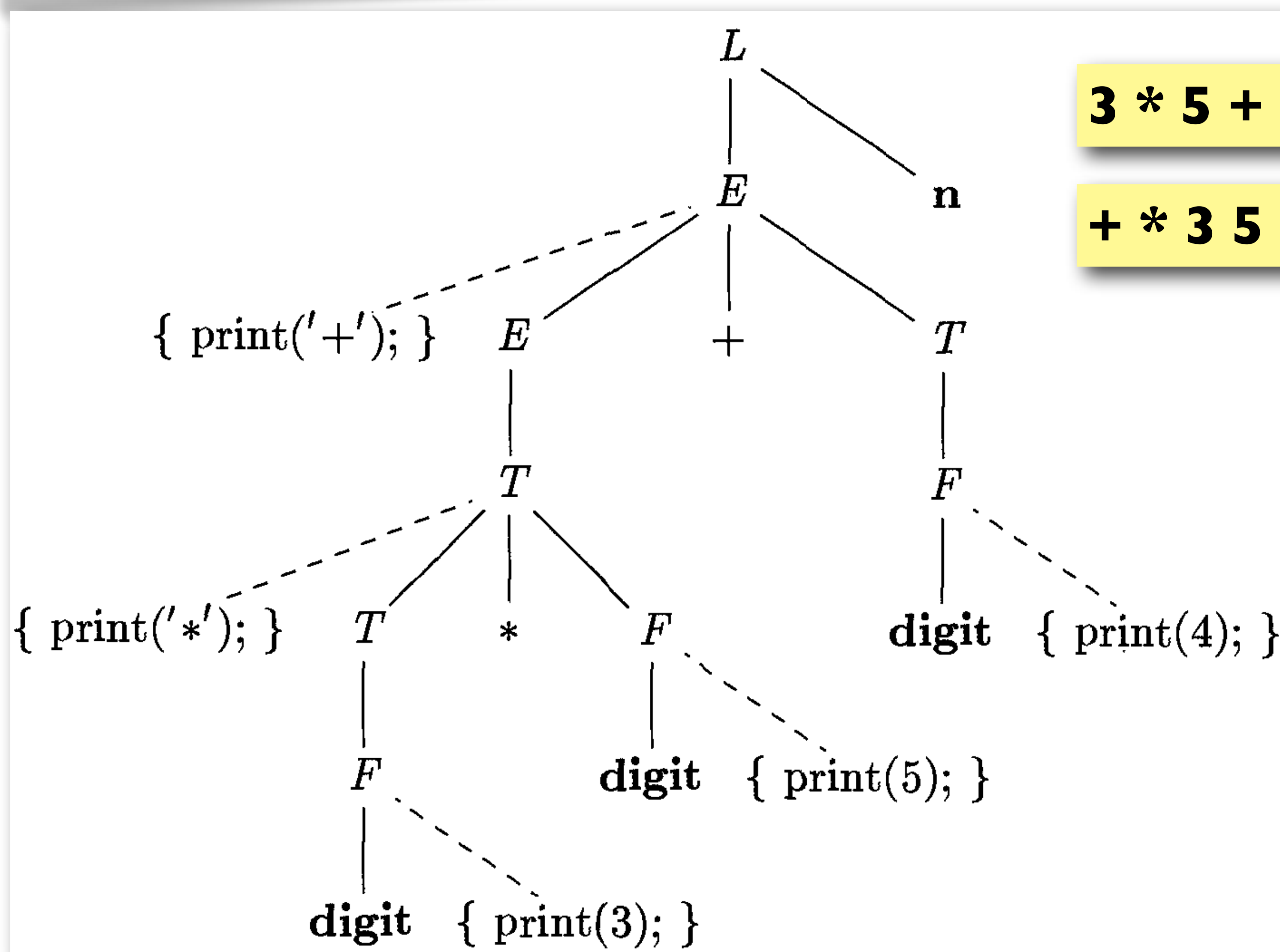
we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\quad E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\quad T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\quad \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

Unfortunately, it is impossible to implement this SDT during either topdown or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of * or +, long before it knows whether these symbols will appear in its input

Any SDT can be implemented as follows:

1. Ignoring the actions, **parse the input and produce a parse tree** as a result.
2. Then, examine each interior node *N*, say one for production A → α (α=β{**a**}δ) Add additional children to *N* for the actions in α, so the children of *N* from left to right have exactly the symbols and actions **a** of α.
3. Perform a preorder traversal (see Section 2.3.4) of the tree, and as soon as a node labeled by an action is visited, perform that action.



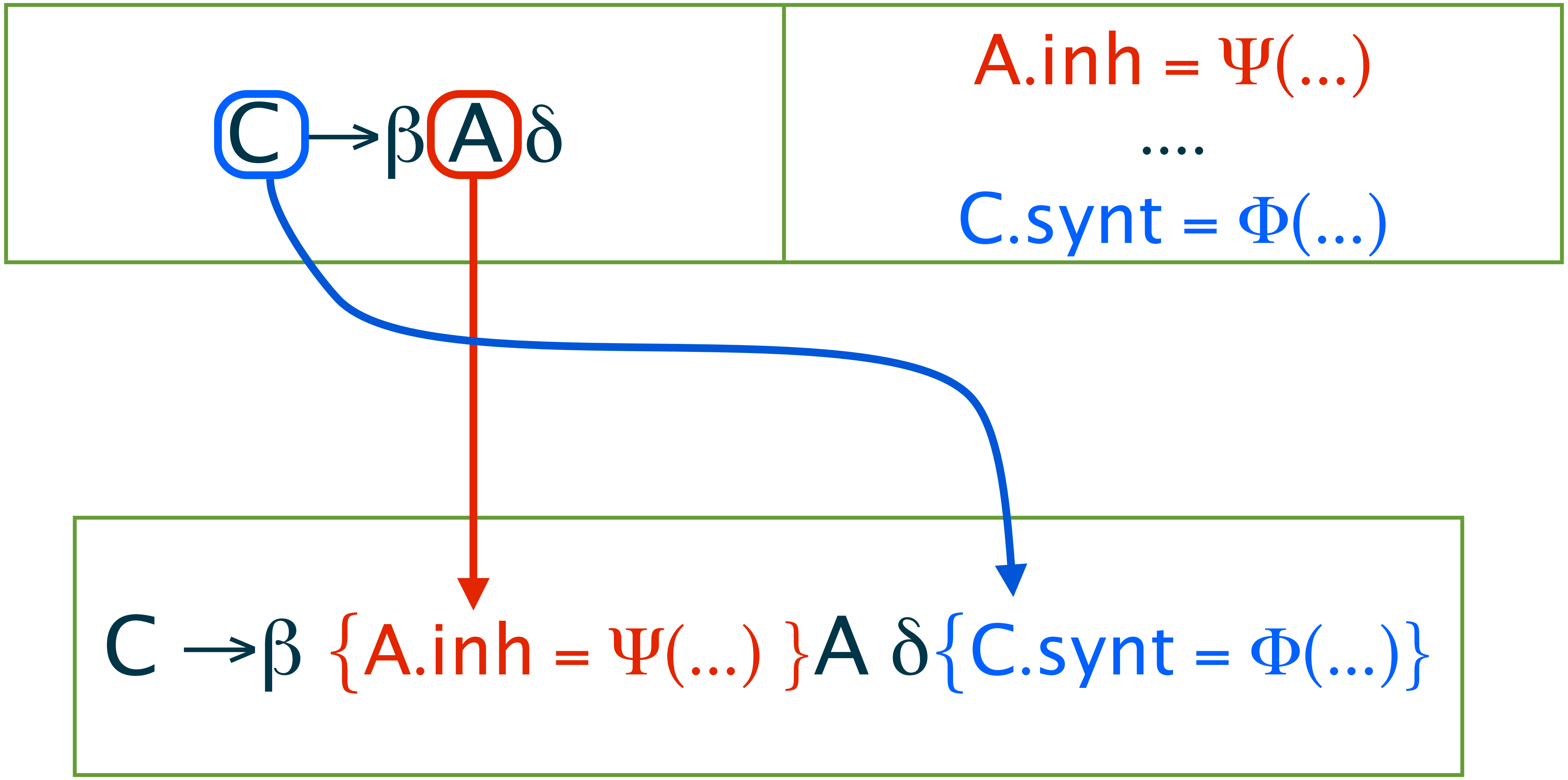
3 * 5 + 4

+ * 3 5 4

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \mathrm{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \mathrm{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \mathrm{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

SDT's for L-Attributed Definitions

If the underlying grammar is not LL(k) it is frequently impossible to perform the translation in connection with either an LL or an LR parser.



The rules for turning an L-attributed SDD into an SDT are as follows:

1. **Embed the action that computes the inherited attributes for a nonterminal *A* immediately before that occurrence of *A* in the body of the production. If several inherited attributes for *A* depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.**

2. **Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.**

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

Exercise:
turn the L-attributed SDD into an SDT

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

D → *T {L.inh:= T .type} L*

T → **int** *{T .type :=integer}*

T → **float** *{T .type :=float}*

L → *{L1 .inh:= L.in} L1 , id {addtype(id.entry, L.inh)}*

L → *id {addType(***id***.entry, L.inh)}*

Build the parse-tree with semantic actions for
real id1 , id2 , id3

D → *T {L.inh:= T .type} L*

T → **int** *{T .type :=integer}*

T → **float** *{T .type :=float}*

L → *{L1 .inh:= L.in} L1 , id {addtype(id.entry, L.inh)}*

L → *id {addType(***id***.entry, L.inh)}*

Build the parse-tree with semantic actions for
real id1 , id2 , id3

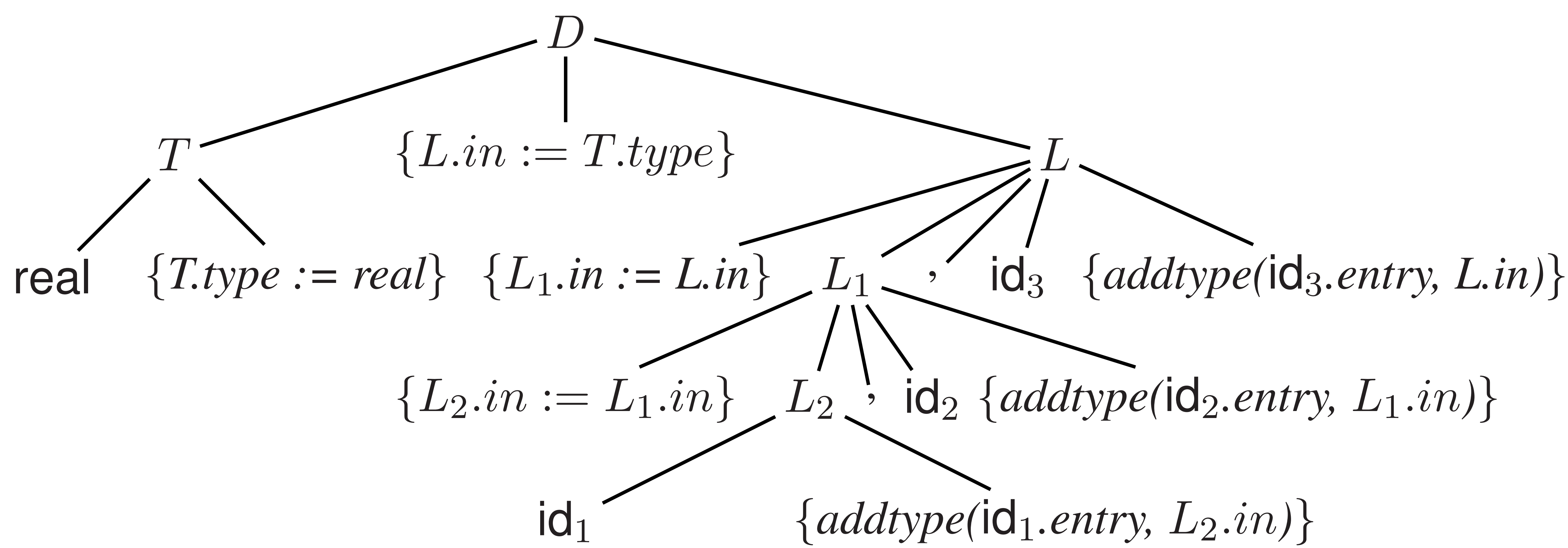
D → *T {L.inh:= T .type} L*

T → **int** *{T .type :=integer}*

T → **float** *{T .type :=float}*

L → *{L1 .inh:= L.in} L1 , id {addtype(id.entry, L.inh)}*

L → *id {addType(***id***.entry, L.inh)}*



Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.

- **When the semantic action involves only synthesized attributes the action can be put at the end of the production.**

IMPLEMENTING L-ATTRIBUTED SDD's

1. ***Build the parse tree and annotate.*** **This method works for any noncircular SDD whatsoever.**

2. ***Build the parse tree, add actions, and execute the actions in preorder.***

3. *Use a recursive-descent parser* with one function for each nonterminal.

   The function for nonterminal *A* receives the inherited attributes of *A* as arguments and returns the synthesized attributes of *A*.

4. *Generate code on the fly,* using a recursive-descent parser.

5. *Implement an SDT in conjunction with an LL-parser*. The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

6. ***Implement an SDT in conjunction with an LR-parser.***

This method may be surprising, since the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed. We shall see, however, that if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.