Syntax Directed Translation

Md. Khorshed Alam

CSE, NDUB

Introduction

- A **syntax-directed definition** specifies the values of attributes by associating semantic rules with the grammar productions
- an infix-to-postfix translator might have a production & rule

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

- 02 non-terminals: **E , T**
- subscript in $\mathbf{E_1}$ distinguishes the occurrence of **E** in the production body from the occurrence of E as the head;
- Both **E** & **T** have a string-valued attribute *code*.
- The semantic rule specifies that the string **E.***code* is formed by concatenating $\mathbf{E_1}$ **.** *code*, **T**. *code*, & the character ' **+**'.
- While the rule makes it explicit that the translation of **E** is built up from the translations of $\mathbf{E_1}$ **, T,** & '**+**', it may be inefficient to implement the translation directly by manipulating strings.

- a **syntax-directed translation** scheme embeds program fragments called *semantic actions within production bodies*, as in

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

- By convention, semantic actions are enclosed within curly braces.
- (If curly braces occur as grammar symbols, we enclose them within single quotes, as in '{' and '}'.)
- The position of a semantic action in a production body determines the order in which the action is executed.
- In above production , the action occurs at the end, after all the grammar symbols;
- in general, semantic actions may occur at any position in a production body.

Which one is to be Chosen Syntax Directed Definition or Syntax Directed Translation?

- Syntax-directed definitions:
  - can be more readable
  - hence more useful for specifications.
- Translation schemes
  - can be more efficient
  - hence more useful for implementations.

Approaches

- **Syntax-directed translation**
- construct a parse tree or a syntax tree , and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree.
- **Translation Schemes**
- translation can be done during parsing, without building an explicit tree.
- "**L-attributed translations**" (L for left-to-right), which encompass virtually all translations that can be performed during parsing.
- "**S-attributed translations**" (S for synthesized) , which can be performed easily in connection with a bottom-up parse.

Syntax-Directed Definitions (SDD)

- is a CFG together with attributes & rules.
- Attributes are associated with grammar symbols & rules are associated with productions.

- If X is a symbol and a is one of its attributes, then we write **X**.*a* to denote the value of *a* at a particular parse-tree node labeled **X**.
- If we implement the nodes of the parse tree by records or objects, then the attributes of **X** can be implemented by **data fields** in the records that represent the nodes for **X**.
- Attributes may be of any kind: numbers , types , table references , or strings, for instance.
- The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

Inherited vs. Synthesized Attributes

1. A **synthesized attribute** for a non-terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.

- Note that the production must have A as its head.
- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

Inherited vs. Synthesized Attributes

2. An **inherited attribute** for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.

- Note that the production must have B as a symbol in its body.
- An inherited attribute at node N is defined only in terms of attribute values at N's parent , N itself, and N's siblings.

While we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.

- **For Non-terminals: S-attribute + L-attribute**
- **For Terminals: S-attribute**

Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Example 5.1 : The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators + and $*$. It evaluates expressions terminated by an endmarker **n**. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

- Production 1: **L** $\rightarrow$ **E n**, sets L. *val* to E. *val*, which we shall see is the numerical value of the entire expression.
- Production 2: **E** $\rightarrow$ $\mathbf{E_1}$ **+ T**, also has one rule, which computes the *val* attribute for the head **E** as the *sum of the values at* E_1 *and T*.

➢ At any parse tree node **N** labeled **E**, the value of *val* for **E** is the sum of the values of *val* at the children of node **N** labeled **E and T**.

- Production 3: **E** $\rightarrow$ **T**, has a single rule that defines the value of *val* for **E** to be the same as the value of *val* at the child for **T**.
- Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them.
- The rules for productions 5 and 6 copy values at a child, like that for the 3rd production.
- Production 7 gives **F**. *val* the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned.

- SDD involves only synthesized attributes (**S-attributed)**;
- In an S-attributed SDD, each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production.

Evaluating an SDD at the Nodes of a Parse Tree

- Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.

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

- How do we construct an annotated parse tree?
- In what order do we evaluate attributes?

❑ Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends

- if all attributes are synthesized, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.

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

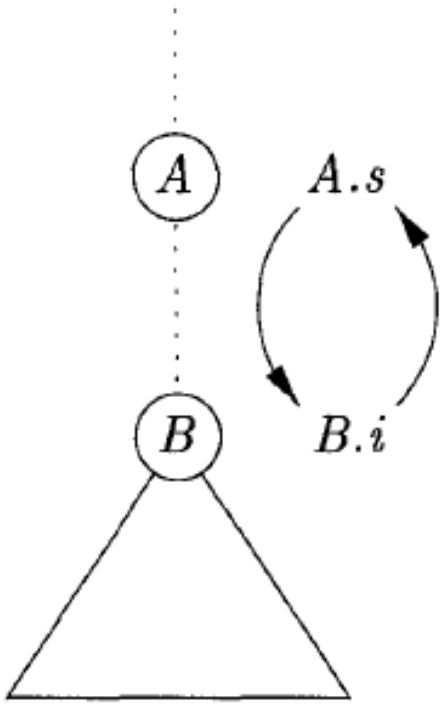
❑For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes.

- consider non-terminals A & B, with synthesized & inherited attributes **A.*s*** & **B.*i***, respectively, along with the production & rules

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

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 $\mathbf{A}_{.s}$ at a node N or $\mathbf{B}_{.i}$ at the child of N without first evaluating the other.
- The circular dependency of $\mathbf{A}_{.s}$ & $\mathbf{B}_{.i}$ at some pair of nodes in a parse tree is suggested by Fig.



Example 5.2 : Figure 5.3 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Fig. 5.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15. □

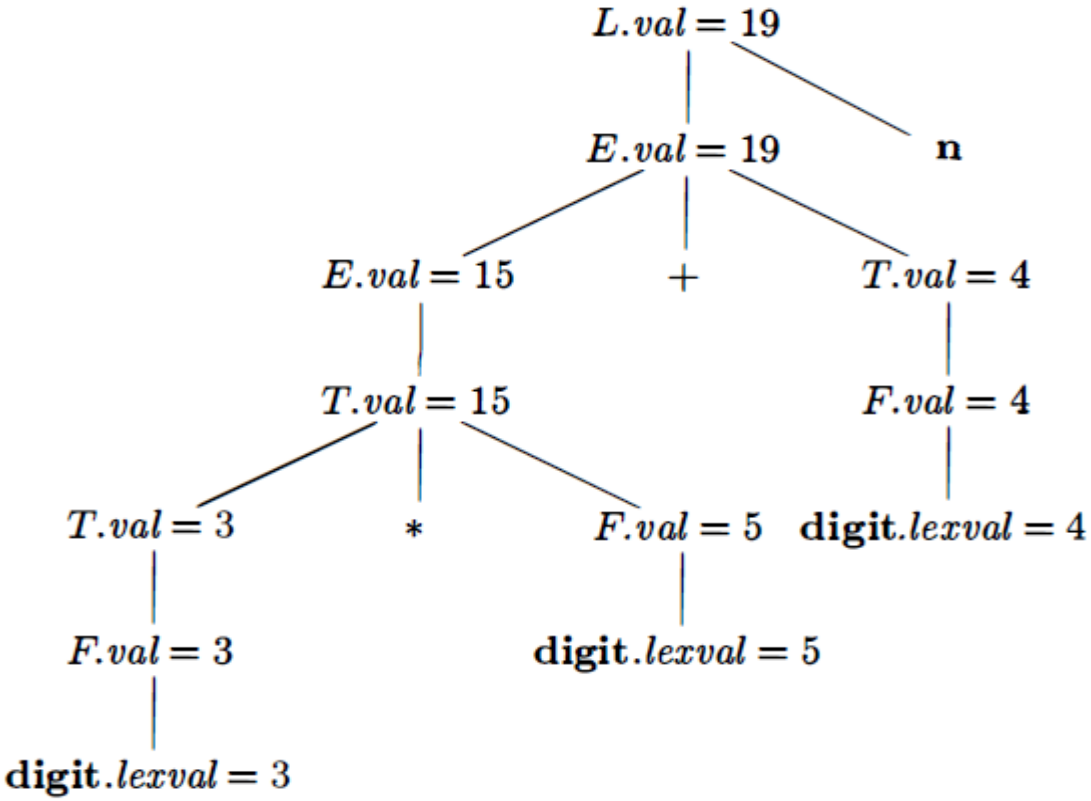


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

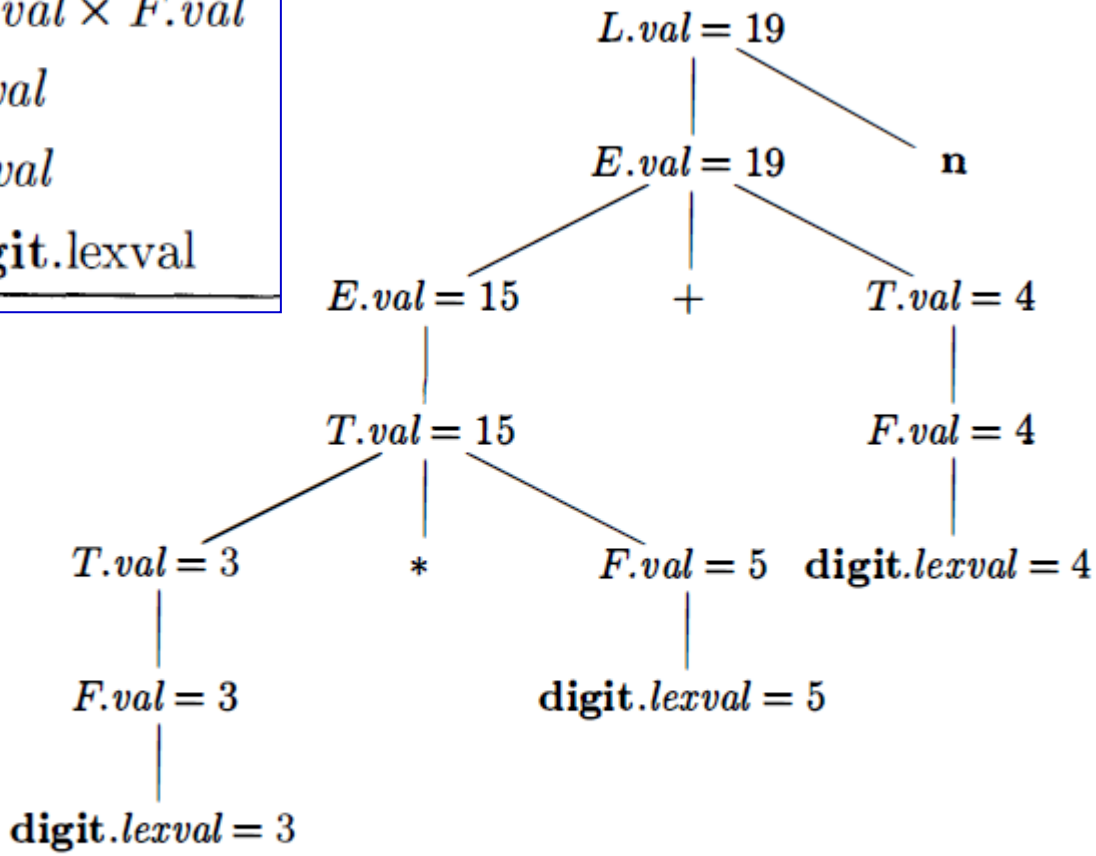


Figure 5.3: Annotated parse tree for $3 * 5 + 4\ \mathbf{n}$

- Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code.

Example 5.3: The SDD in Fig. 5.4 computes terms like $3 * 5$ and $3 * 5 * 7$. The top-down parse of input $3 * 5$ begins with the production $T \rightarrow F\,T'$. Here, F generates the digit 3, but the operator $*$ is generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from $*$. An inherited attribute will therefore be used to pass the operand to the operator.

The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar; we used such a grammar as a running example to illustrate top-down parsing in Section 4.4.

	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)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Figure 5.4: 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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

- Each of the non-terminals **T** & **F** has a synthesized attribute *val*;
- The terminal **digit** has a synthesized attribute *lexval*.
- The non-terminal **T'** has two attributes:

✓ an inherited attribute *inh*

✓ a synthesized attribute *syn*.

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. Given a term $x * y * z$, the root of the subtree for $* y * z$ inherits x. Then, the root of the subtree for $* z$ inherits the value of $x * y$, and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

	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)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

T`.inh=F.val

- The production at the node for **T'** is **T' →* FT$_1$`** .The inherited attribute **T$_1$`.inh** is defined by the semantic rule **T$_1$`.inh = T'.inh x F.val**

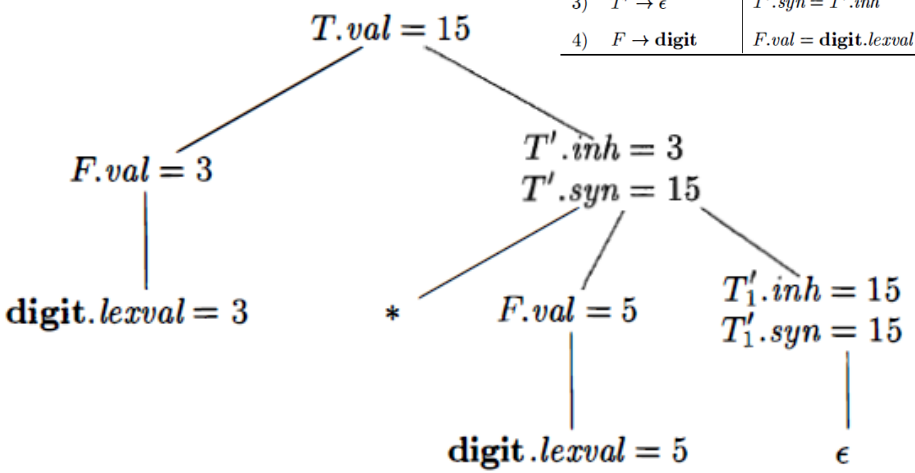


Figure 5.5: Annotated parse tree for $3 * 5$

With **T'.inh = 3** & **F.val = 5**, we get **T$_1$`.inh = 15**. At the lower node for **T$_1$`**, the production **T' →ε** The rule **T'.syn = T'.inh** defines **T$_1$`.syn = 15**.

- The *syn* attributes at the nodes for **T'** pass the value 15 up the tree to the node for T, where **T.val = 15**.

Evaluation Orders for SDD

- "**Dependency graphs**" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.
- Two important classes of SDD's:

❑ **S-attributed**

❑ **L-attributed**

Dependency Graphs

- A **dependency graph** depicts the flow of information among the attribute instances in a particular parse tree; *an edge from one attribute instance to another means that the value of the first is needed to compute the second*.
- Edges express constraints implied by the semantic rules.

Properties

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
- 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 $A.b$. More precisely, at every node N labeled A where production p is applied, create an edge to attribute b at N, from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production.[2]

- 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 $B.c$. For each node N labeled B that corresponds to an occurrence of this B in the body of production p, create an edge to attribute c at N from the attribute a at the node M that corresponds to this occurrence of X. Note that M could be either the parent or a sibling of N .

Example 5.4: Consider the following production and rule:

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

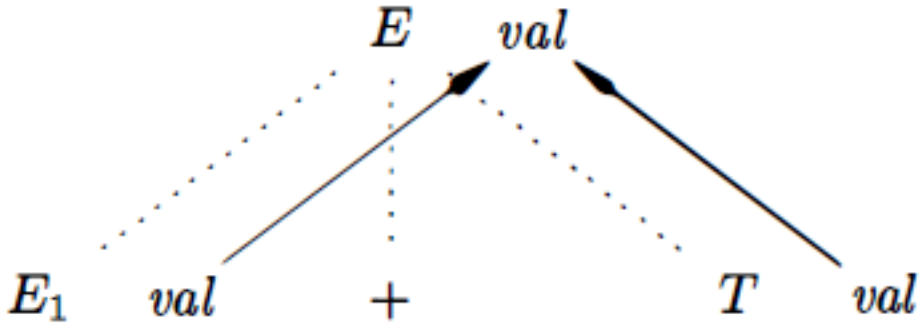
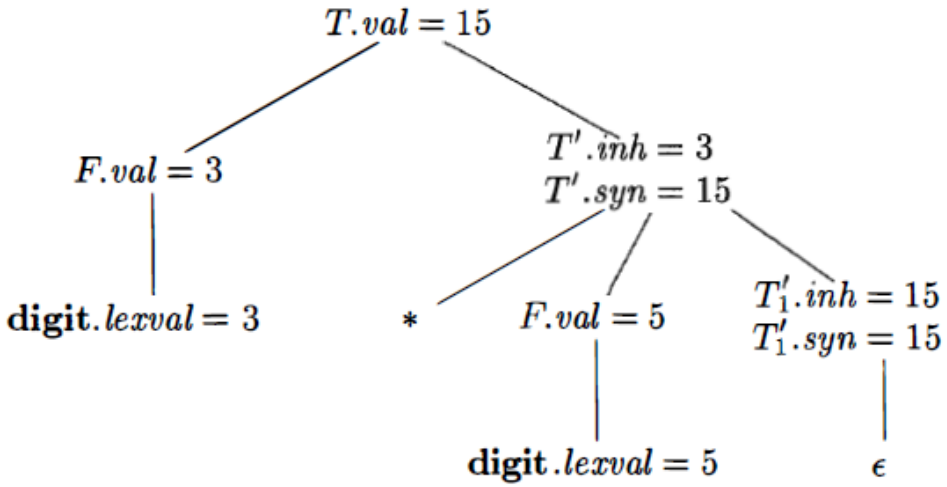
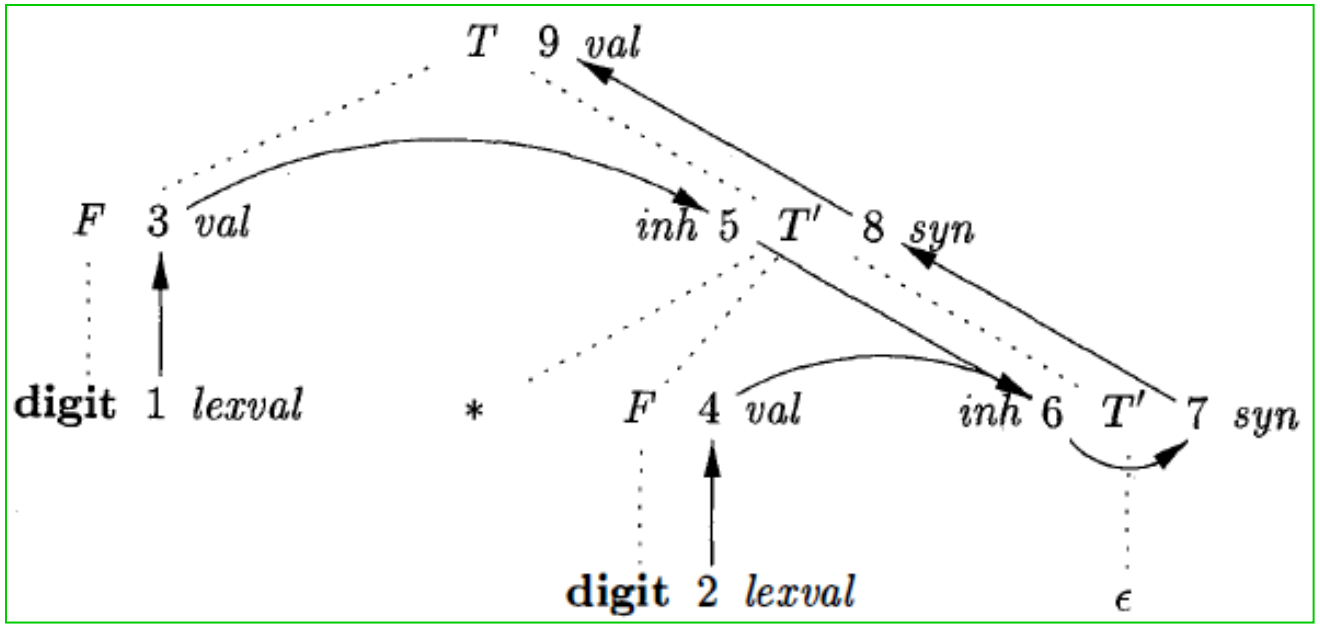


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Example 5.5: An example of a complete dependency graph appears in Fig. 5.7. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.



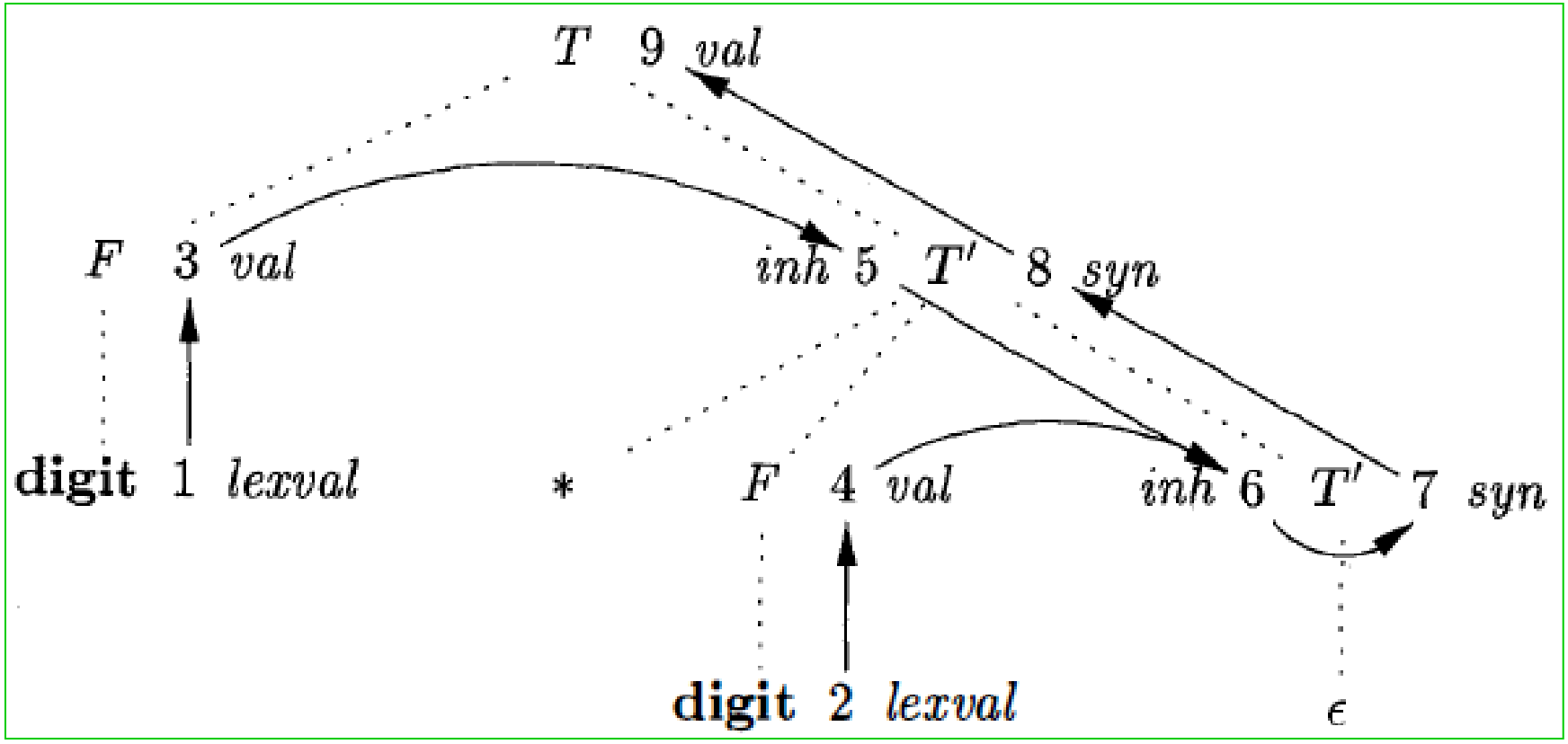
	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$



Ordering the Evaluation of Attributes

- The dependency graph characterizes the *possible orders in which we can evaluate the attributes at the various nodes* of a parse tree.
- 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*.
- Thus, the only allowable orders of evaluation are those sequences of nodes $\mathbf{N_1}$, $\mathbf{N_2}$, …. , $\mathbf{N_k}$ 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.

Example 5.6: The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: $1, 2, \ldots, 9$. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as $1, 3, 5, 2, 4, 6, 7, 8, 9$. □



S-Attributes Definitions

- An SDD is S-attributed if every attribute is synthesized

Example:

S-Attributes:

$L_{.val}$, $E_{.val}$, $T_{.val}$, $F_{.val}$

We can evaluate its attributes in any bottom up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree & evaluating the attributes at a node N when the traversal leaves N for the last time.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

L-Attributes Definitions

- 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") .

❑ **Characteristics(each attribute must be either):**

1. Synthesized, or
2. Inherited, but with the rules limited as follows . Suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$, and that 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) Either 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

Example 5.8 : The SDD in Fig. 5.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\, T'$	$T'.inh = F.val$
$T' \rightarrow * F\, T'_1$	$T'_1.inh = T'.inh \times F.val$

- The first of these rules defines the inherited attribute **T'.inh** using only **F.val**, & F appears to the left of T' in the production body
- The second rule defines **T`1.inh** using the inherited attribute **T'.inh** associated with the head, and **F.val**, where F appears to the left of **T`1.inh** in the production body.
- In each of these cases, the rules use information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.

Example 5.9: Any SDD containing the following production and rules cannot be L-attributed:

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

Semantic Rules with Controlled Side Effects

- Translations involve side effect s: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table.
- 02 Ways of controlling side effects in SDD's:

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

As an example of an incidental side effect, let us modify the desk calculator of Example 5.1 to print a result. Instead of the rule $L.val = E.val$, which saves the result in the synthesized attribute $L.val$, consider:

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E\ \mathbf{n}$	$print(E.val)$

Semantic rules that are executed for their side effects, such as $print(E.val)$, will be treated as the definitions of dummy synthesized attributes associated with the head of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into $E.val$.

Example 5.10: The SDD in Fig. 5.8 takes a simple declaration D consisting of a basic type T followed by a list L of identifiers. T can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

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

Figure 5.8: Syntax-directed definition for simple type declarations

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

Nonterminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, $T.type$, which is the type in the declaration D. Nonterminal L also has one attribute, which we call *inh* to emphasize that it is an inherited attribute. The purpose of $L.inh$ is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

Productions 2 and 3 each evaluate the synthesized attribute $T.type$, giving it the appropriate value, integer or float. This type is passed to the attribute $L.inh$ in the rule for production 1. Production 4 passes $L.inh$ down the parse tree. That is, the value $L_1.inh$ is computed at a parse-tree node by copying the value of $L.inh$ from the parent of that node; the parent corresponds to the head of the production.

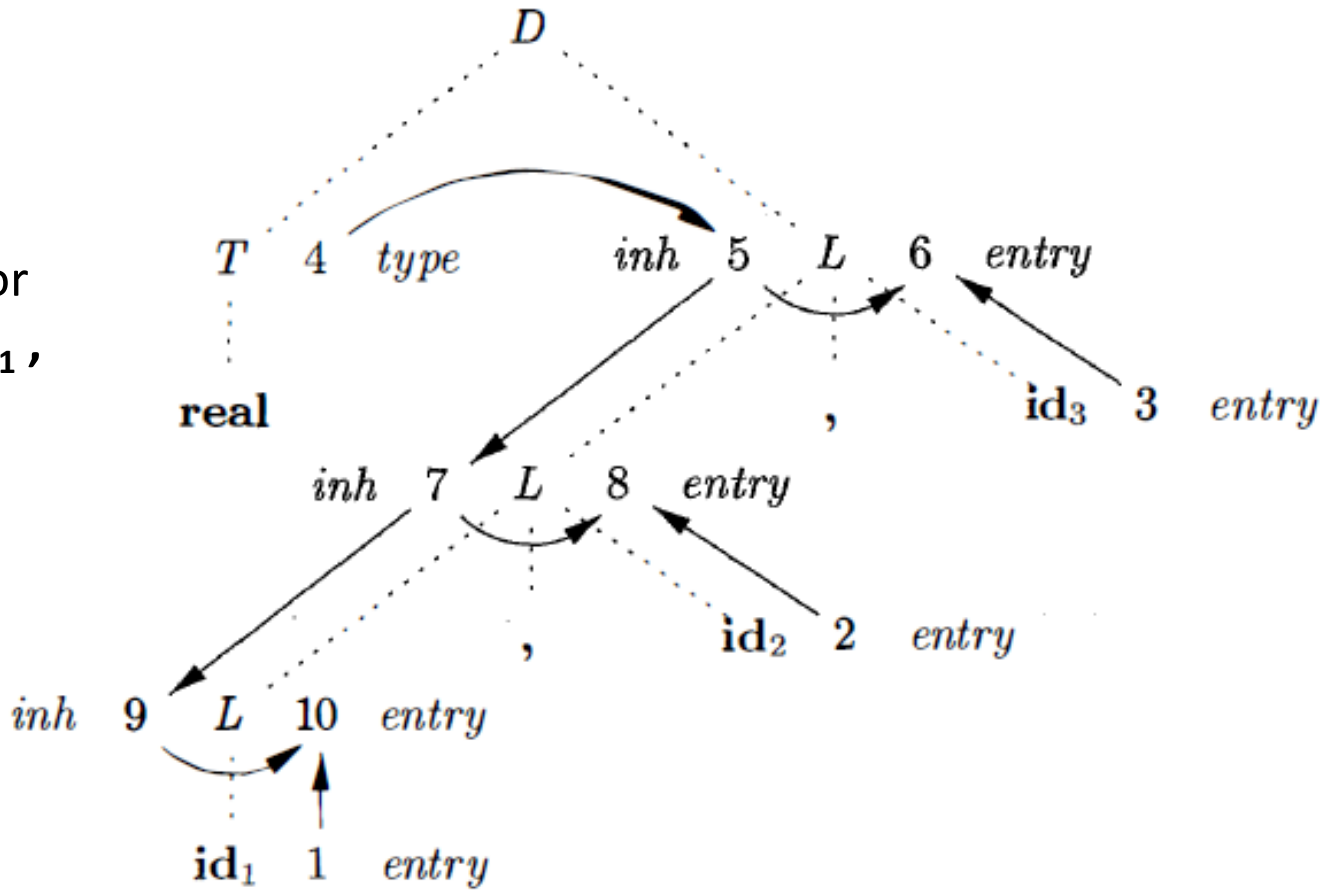
	$addType(\mathbf{id}.entry, L.inh)$
5) $L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

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.

Figure 5.9:
Dependency graph for a declaration **float id_1 , id_2 , id3**



Nodes 1, 2, and 3 represent the attribute *entry* associated with each of the leaves labeled **id**. Nodes 6, 8, and 10 are the dummy attributes that represent the application of the function *addType* to a type and one of these *entry* values.

Node 4 represents the attribute *T.type*, and is actually where attribute evaluation begins. This type is then passed to nodes 5, 7, and 9 representing *L.inh* associated with each of the occurrences of the nonterminal *L*. □

Application of Syntax-Directed Translation

- SDT techniques will be applied to type checking & intermediate-code generation.
- **The main application is the construction of syntax trees**.
- To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.
- S-attributed definition, is suitable for use during bottom-up parsing.
- L-attributed, is suitable for use during top-down parsing.

Construction of Syntax Tree

- Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
- A syntax-tree node representing an expression $E_1 + E_2$
  - has label +
  - 02 children representing the sub expressions E_1 and E_2

- We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an ***op*** field that is the label of the node.
- **Additional fields :**

1. 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.
2. 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_k .

Example 5.11: The S-attributed definition in Fig. 5.10 constructs syntax trees for a simple expression grammar involving only the binary operators + and −. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

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

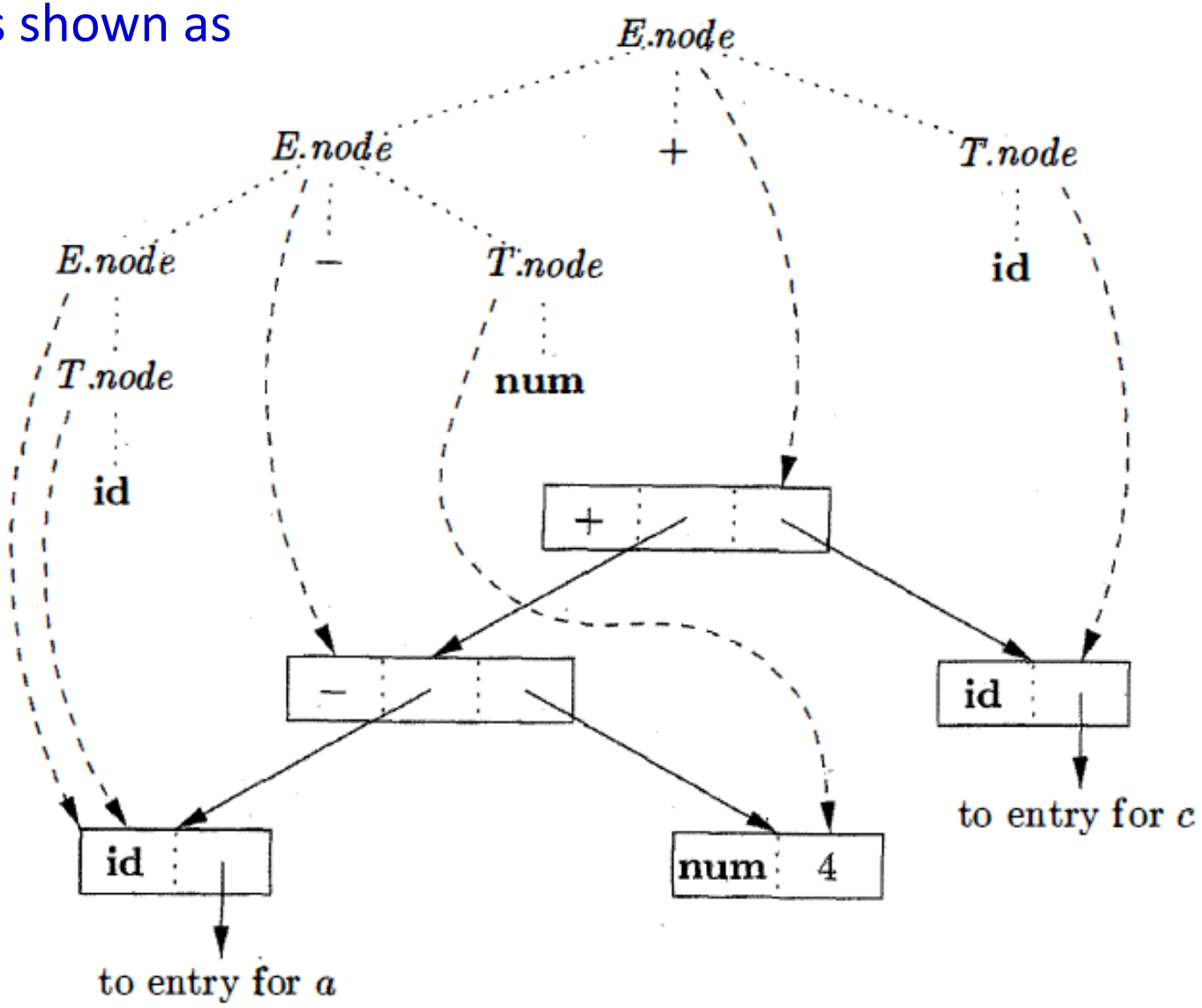
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node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

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with $'+'$ for op and two children, $E_1.node$ and $T.node$, for the subexpressions. The second production has a similar rule.

The last two T-productions have a single terminal on the right. We use the constructor $Leaf$ to create a suitable node, which becomes the value of $T.node$.

Syntax tree for a-4+c

Syntax-tree edges shown as solid lines.



- If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps:

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) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

P5: pointing to the root of the constructed syntax tree.

Syntax-Directed Translation Schemes

- SDT is a context-free grammar with program fragments embedded within production bodies
- The program fragments are called semantic actions & can appear at any position within a production body.
- By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

Postfix Translation Schemes

- We can construct an SDT in which each action is placed at the end of the production & 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***.

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

L	$\rightarrow$	E **n**	$\{$ 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	$\rightarrow$	**digit**	$\{$ $F.val =$ **digit**$.lexval;$ $\}$

SDT's with Actions Inside Production

- An action may be placed at any position within the body of a production.
- It is performed immediately after all symbols to its left are processed.
- Thus, if we have a production **B → X {a} Y**, the action ***a*** is done after we have recognized **X** (if X is a terminal) or all the terminals derived from X (if X is a non-terminal).

- More precisely,

✓ If the parse is bottom-up, then we perform action ***a*** as soon as this occurrence of X appears on the top of the parsing stack.

✓ If the parse is top-down, we perform ***a*** just before we attempt to expand this occurrence of Y (if Y a non-terminal) or check for Y on the input (if Y is a terminal) .

Example 5.16 : As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 5.21.

1)	L	$\rightarrow$	E **n**
2)	E	$\rightarrow$	{ print('+'); } $E_1 + T$
3)	E	$\rightarrow$	T
4)	T	$\rightarrow$	{ print('*'); } $T_1 * F$
5)	T	$\rightarrow$	F
6)	F	$\rightarrow$	(E)
7)	F	$\rightarrow$	**digit** { print(**digit**.*lexval*); }

1) $L \rightarrow E\ \mathbf{n}$
2) $E \rightarrow \{\ \text{print}('+');\ \}\ E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow \{\ \text{print}('*');\ \}\ T_1 * F$
5) $T \rightarrow F$
6) $F \rightarrow (\ E\ )$
7) $F \rightarrow \mathbf{digit}\ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}$

Unfortunately, it is impossible to implement this SDT during either top-down 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.

Using marker nonterminals M_2 and M_4 for the actions in productions 2 and 4, respectively, on input 3, a shift-reduce parser (see Section 4.5.3) has conflicts between reducing by $M_2 \rightarrow \epsilon$, reducing by $M_4 \rightarrow \epsilon$, and shifting the digit. □

Any SDT can be implemented

1. Ignoring the actions, parse the input & produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \rightarrow \alpha$. Add additional children to N for the actions in α , so the children of N from left to right have exactly the symbols & actions of α
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

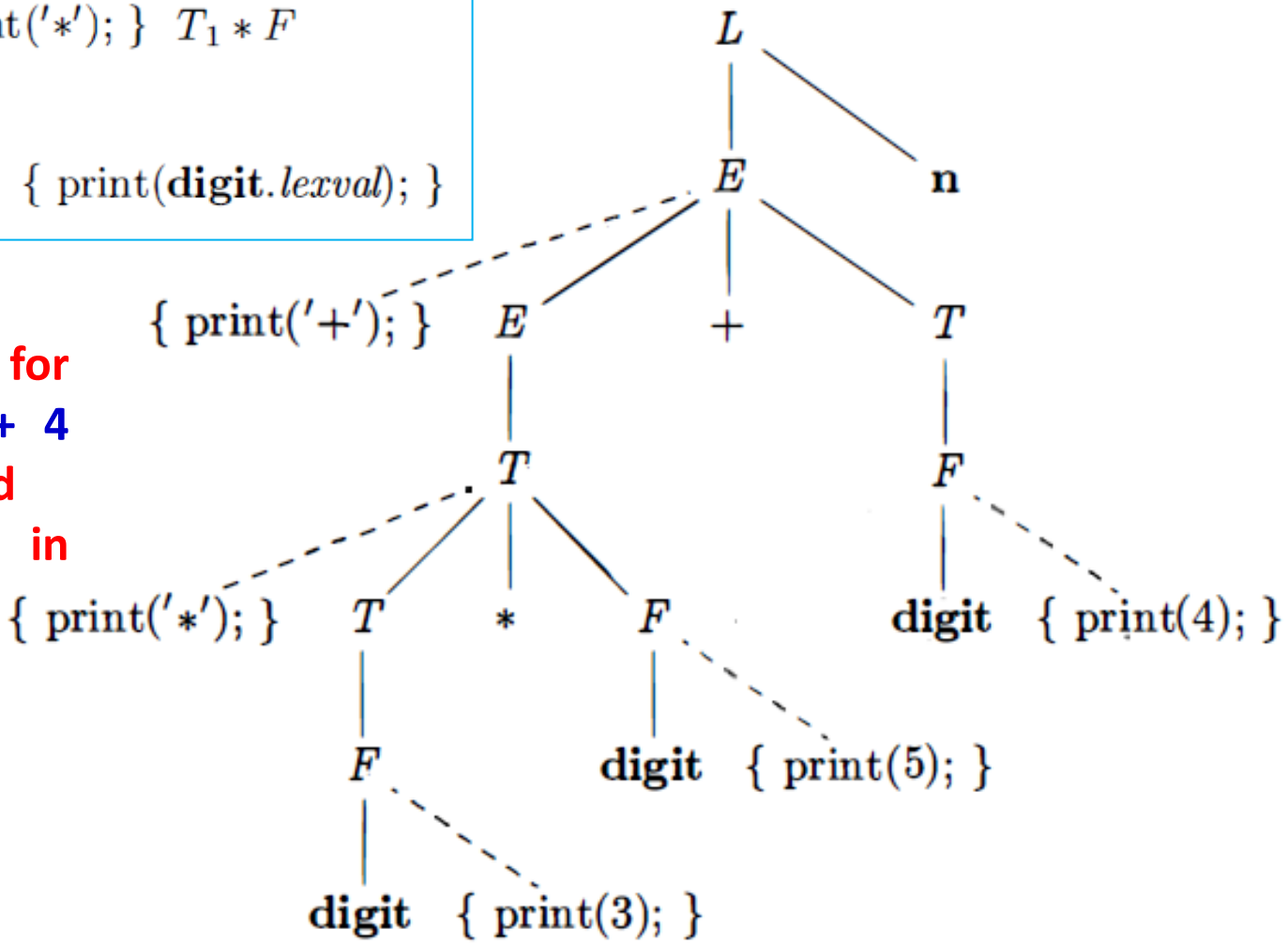
1)	L	$\rightarrow$	E **n**
2)	E	$\rightarrow$	{ print('+'); } $E_1 + T$
3)	E	$\rightarrow$	T
4)	T	$\rightarrow$	{ print('*'); } $T_1 * F$
5)	T	$\rightarrow$	F
6)	F	$\rightarrow$	(E)
7)	F	$\rightarrow$	**digit** { print(**digit**.*lexval*); }

Preorder: action is done when we first Visit a node

- **Parse tree for expression 3 * 5 + 4 with actions inserted**
- **visit the nodes in preorder,**

prefix form

+ * 3 5 4



The preorder of a (sub) tree rooted at node N consists of N, followed by the preorders of the subtrees of each of its children, from the left.

Thanks ☺