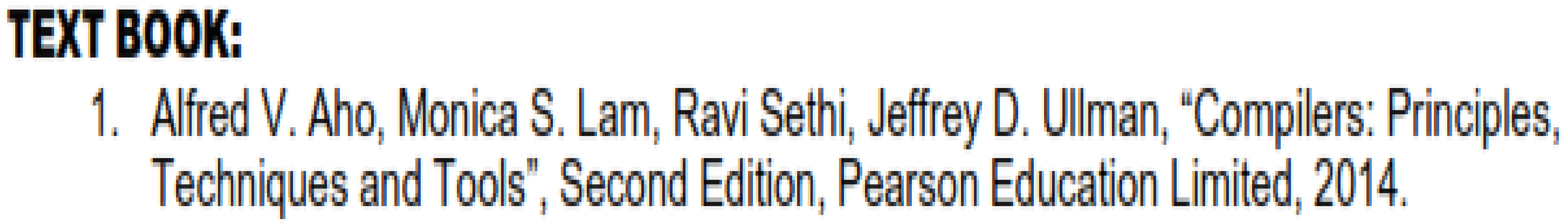
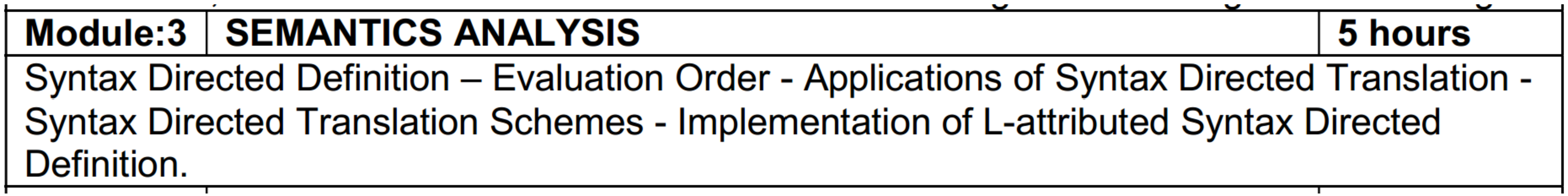
BCSE307L – COMPILER DESIGN



TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.

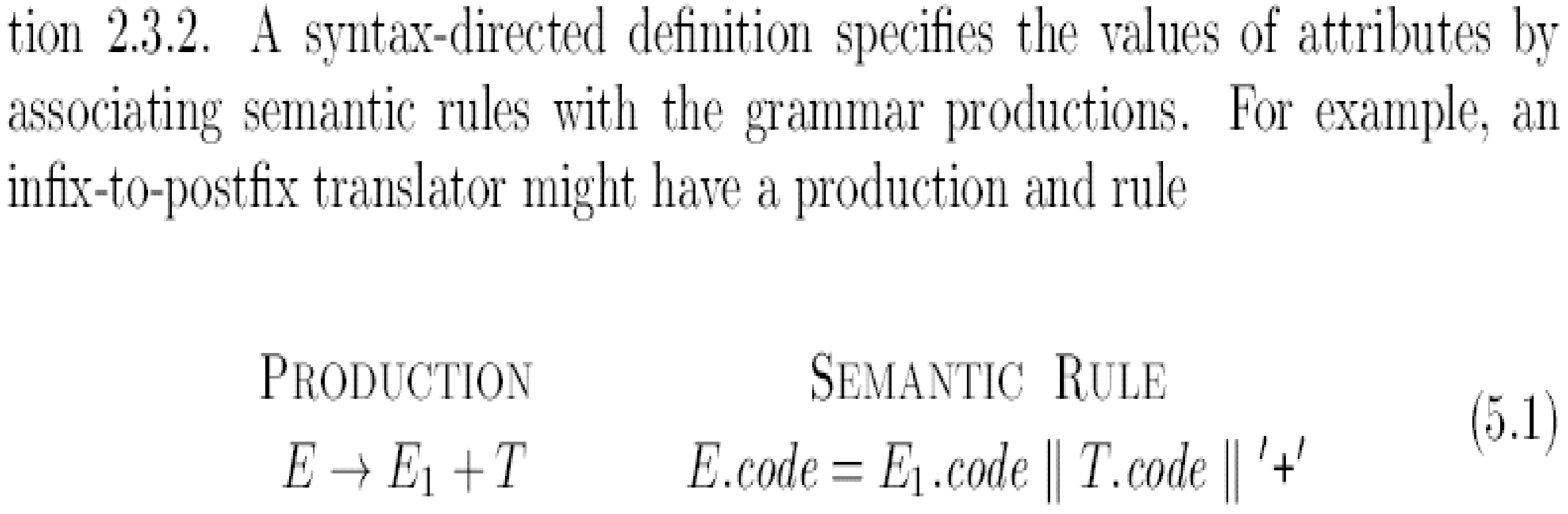


Module:3	SEMANTICS ANALYSIS	5 hours
Syntax Directed Definition – Evaluation Order - Applications of Syntax Directed Translation - Syntax Directed Translation Schemes - Implementation of L-attributed Syntax Directed Definition.		

Syntax-Directed Translation (SDT)

- **Syntax-Directed Definition (SDD)**
- **Syntax-Directed Translation (SDT)**

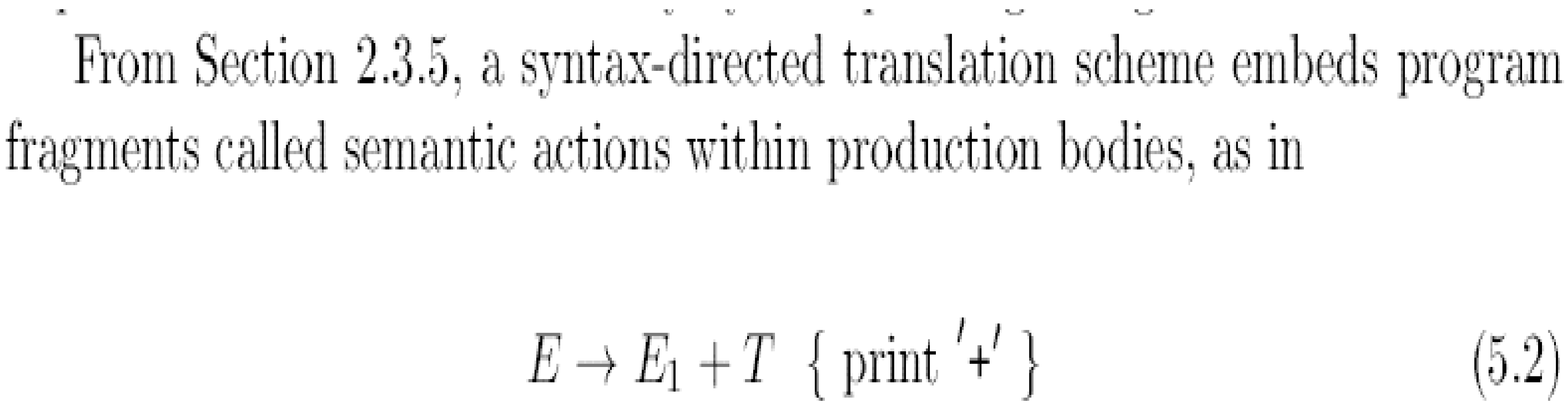
Syntax-Directed Definition (SDD)



tion 2.3.2. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	
$E \to E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$	(5.1)

Syntax-Directed Translation (SDT)



From Section 2.3.5, a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

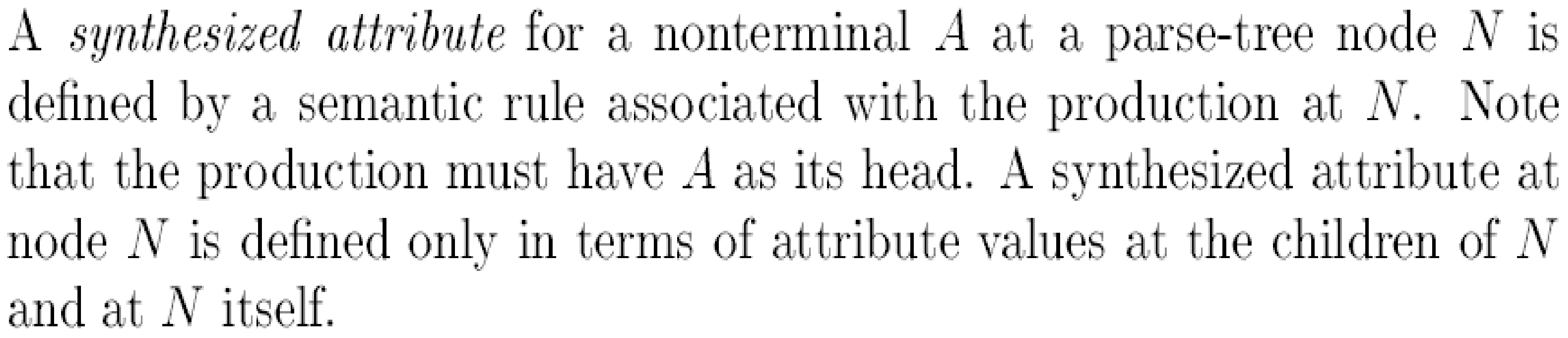
$$E \rightarrow E_1 + T \;\; \{ \text{ print } '+' \; \} \qquad (5.2)$$

Syntax-Directed Definition (SDD)

- Inherited and Synthesized Attributes
  - Synthesized Attribute
  - Inherited Attribute
- Evaluating an SDD at the Nodes of a Parse Tree

Inherited and Synthesized Attributes

- Synthesized Attribute



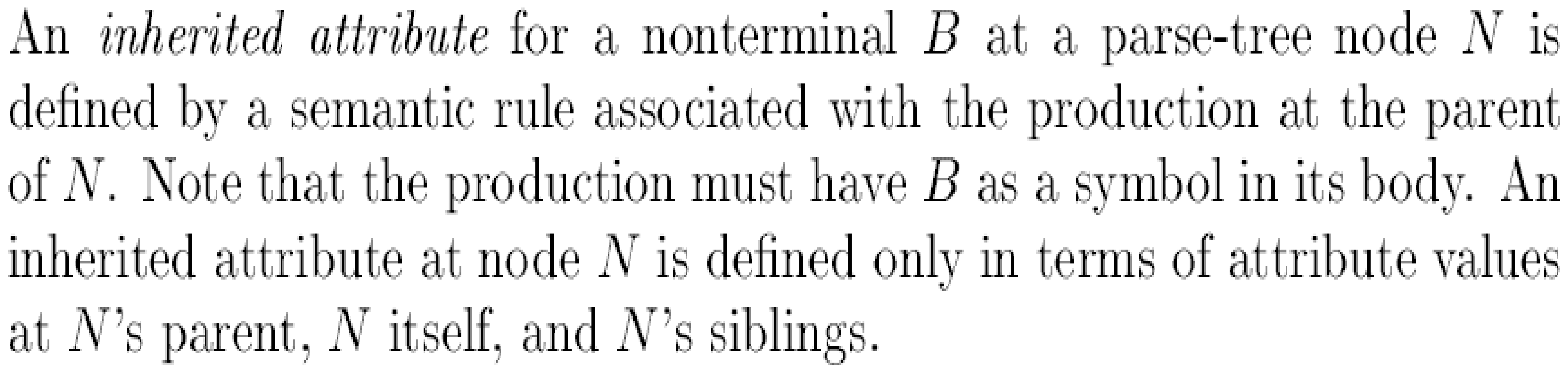
A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

Synthesized attribute

- Parent, itself and Siblings

Inherited and Synthesized Attributes

- Inherited Attribute



An *inherited attribute* for a nonterminal 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.

Inherited attribute

- children and itself

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.

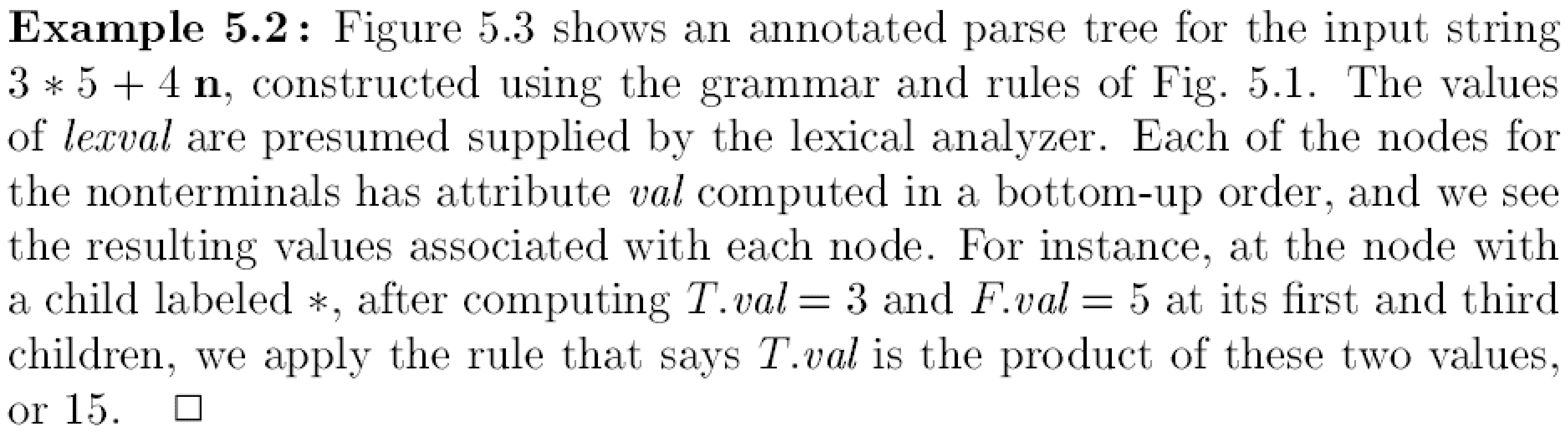
	Production	Semantic Rules
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1\ +\ T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1\ *\ F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (\ E\ )$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\mathrm{lexval}$

Figure 5.1: Syntax-directed definition of a simple desk calculator

Evaluating an SDD at the Nodes of a Parse Tree

- Annotated Parse Tree

Evaluating an SDD at the Nodes of a Parse Tree



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. □

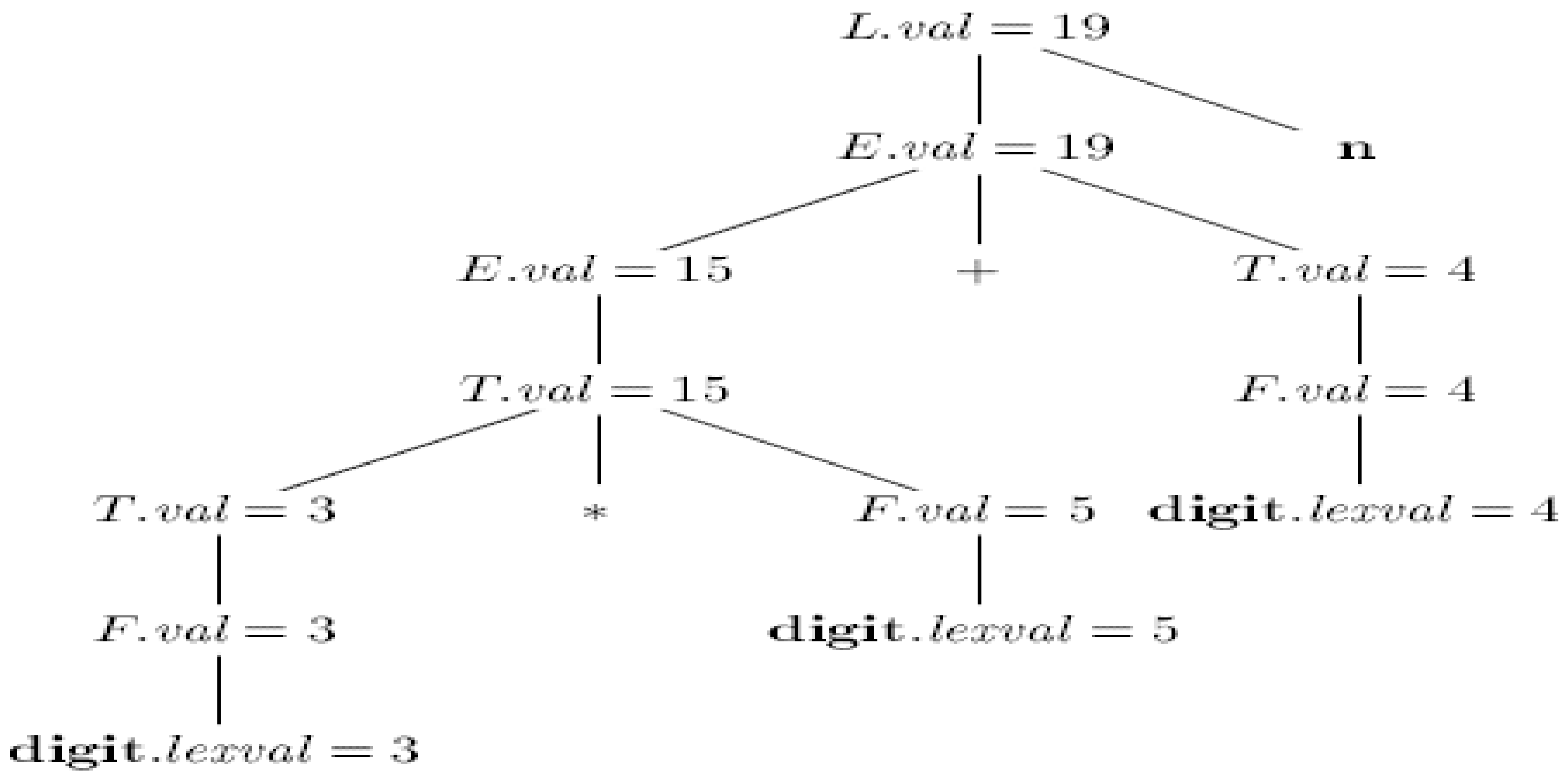
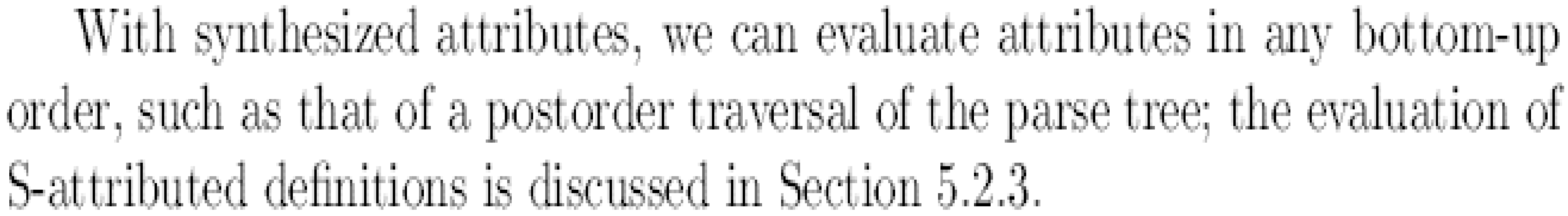


Figure 5.3: Annotated parse tree for $3 * 5 + 4\ \mathbf{n}$

Evaluating an SDD at the Nodes of a Parse Tree



With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree; the evaluation of S-attributed definitions is discussed in Section 5.2.3.

For instance, consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules

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. The circular dependency of $A.s$ and $B.i$ at some pair of nodes in a parse tree is suggested by Fig. 5.2.

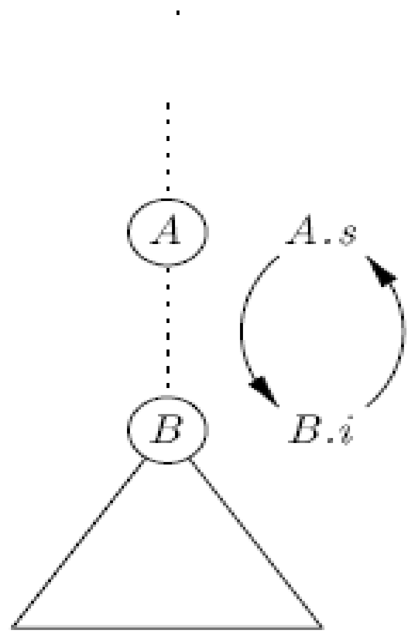


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

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.

	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)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

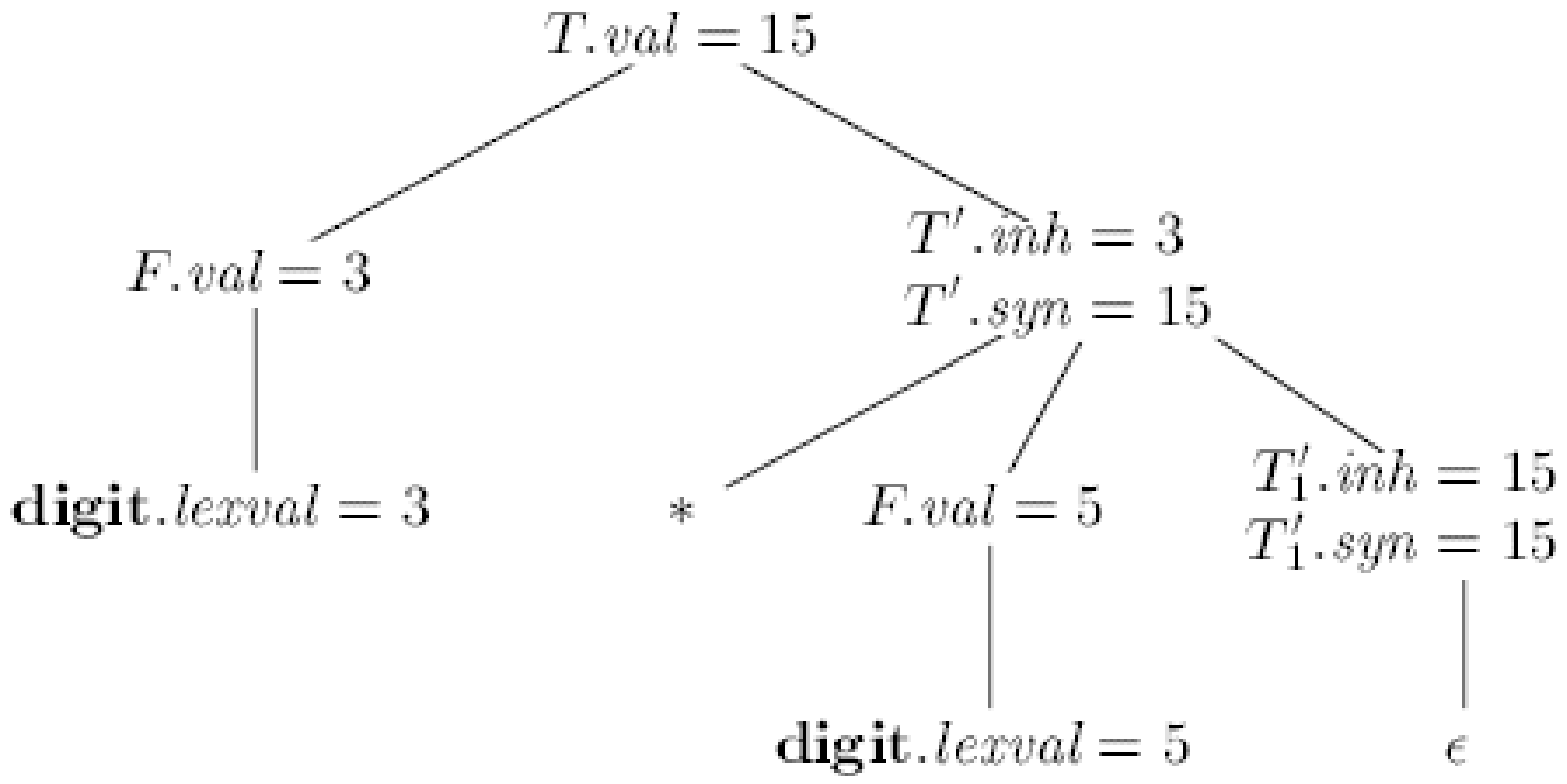
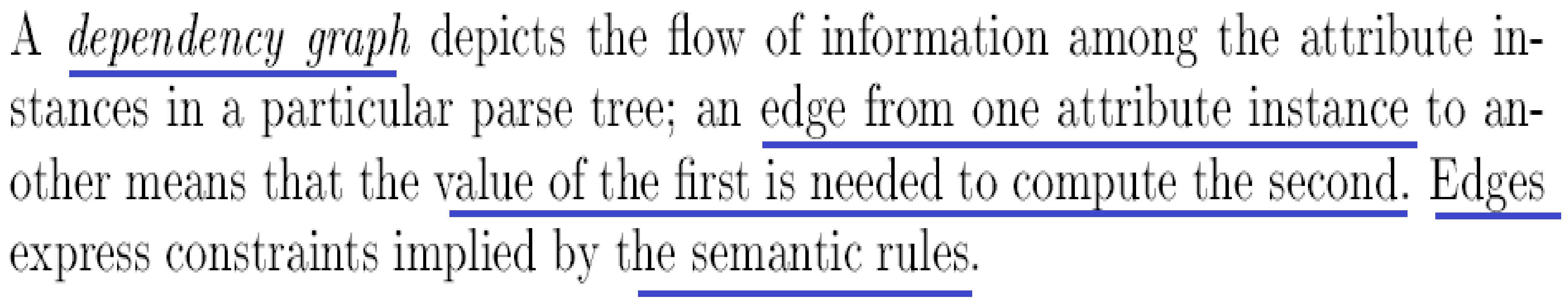


Figure 5.5: Annotated parse tree for $3 * 5$

Evaluation Orders for SDD’s

- Dependency Graph
- Ordering the evaluation of Attributes
- S – Attributed Definitions
- L – Attributed Definitions
- Semantic Rules with Controlled Side Effects

Dependency Graph



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.

- 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.

Dependency Graph

Example 5.4: Consider the following production and rule:

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

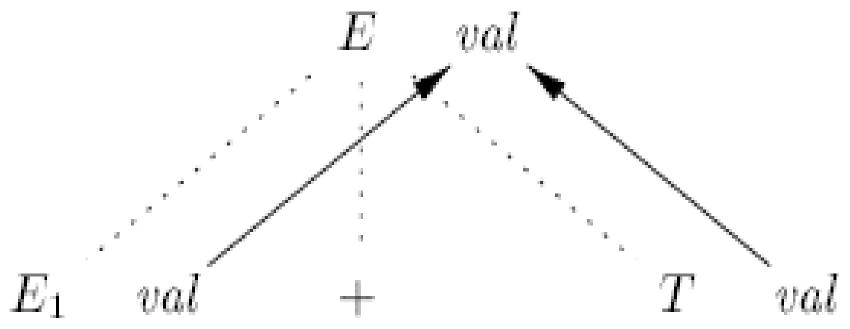


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $T.val$

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.

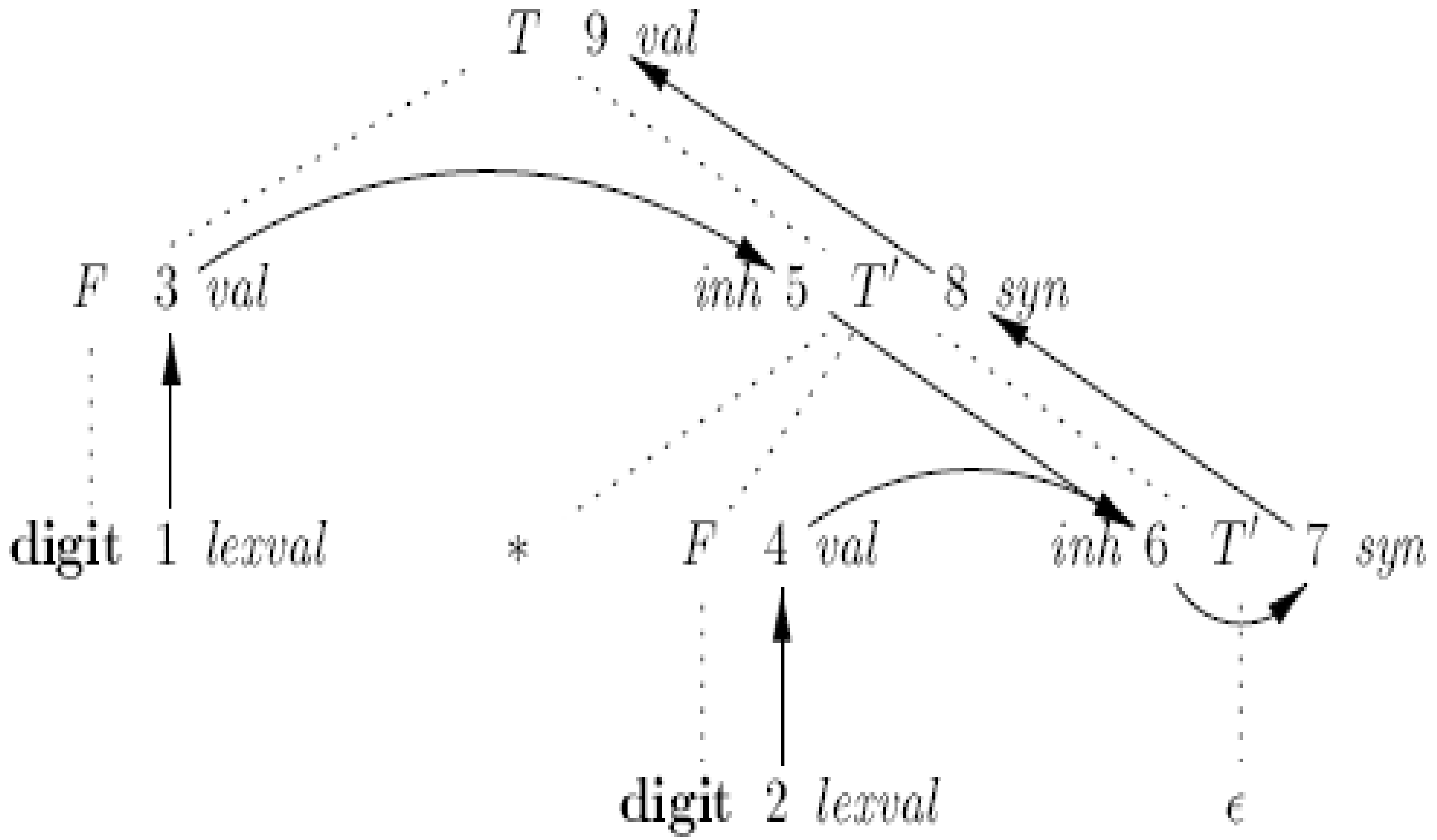


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

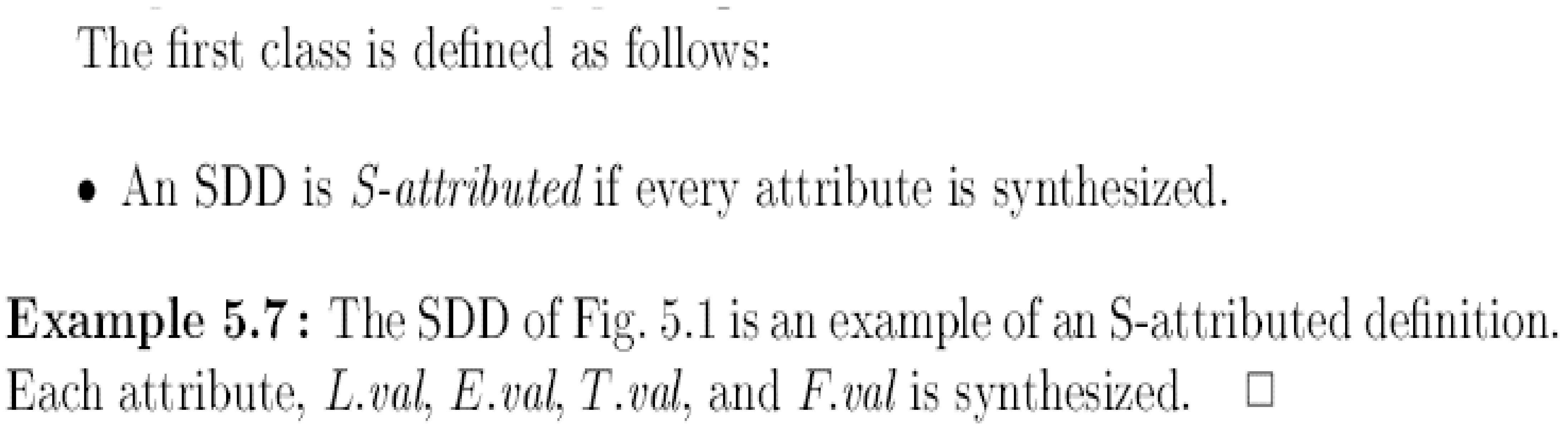
Ordering the evaluation of Attributes

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.

The evaluation orders are those sequences of nodes $N_1, N_2, \ldots, N_k$ 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 – Attributed Definitions



The first class is defined as follows:

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

Example 5.7 : The SDD of Fig. 5.1 is an example of an S-attributed definition. Each attribute, *L.val*, *E.val*, *T.val*, and *F.val* is synthesized. □

S – Attributed Definitions

When an SDD is S-attributed, 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 and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function *postorder*, defined below, to the root of the parse tree (see also the box "Preorder and Postorder Traversals" in Section 2.3.4):

```
postorder(N) {
        for ( each 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.

L – Attributed Definitions

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \to X_1 X_2 \cdots X_n$, 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
$T \rightarrow F\,T'$	$T'.inh = F.val$
$T' \rightarrow * \, F\,T'_1$	$T'_1.inh = T'.inh \times F.val$

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

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

Semantic Rules with Controlled Side Effects

Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment; translation schemes

Control Side effects in SDD's is one of the following ways

permit side effects when attribute evaluation based on any topological sort of the dependency graph

Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order.

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)	$L \to E\ \mathbf{n}$	$print(E.val)$

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.

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 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.

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

Figure 5.8: Syntax-directed definition for simple type declarations

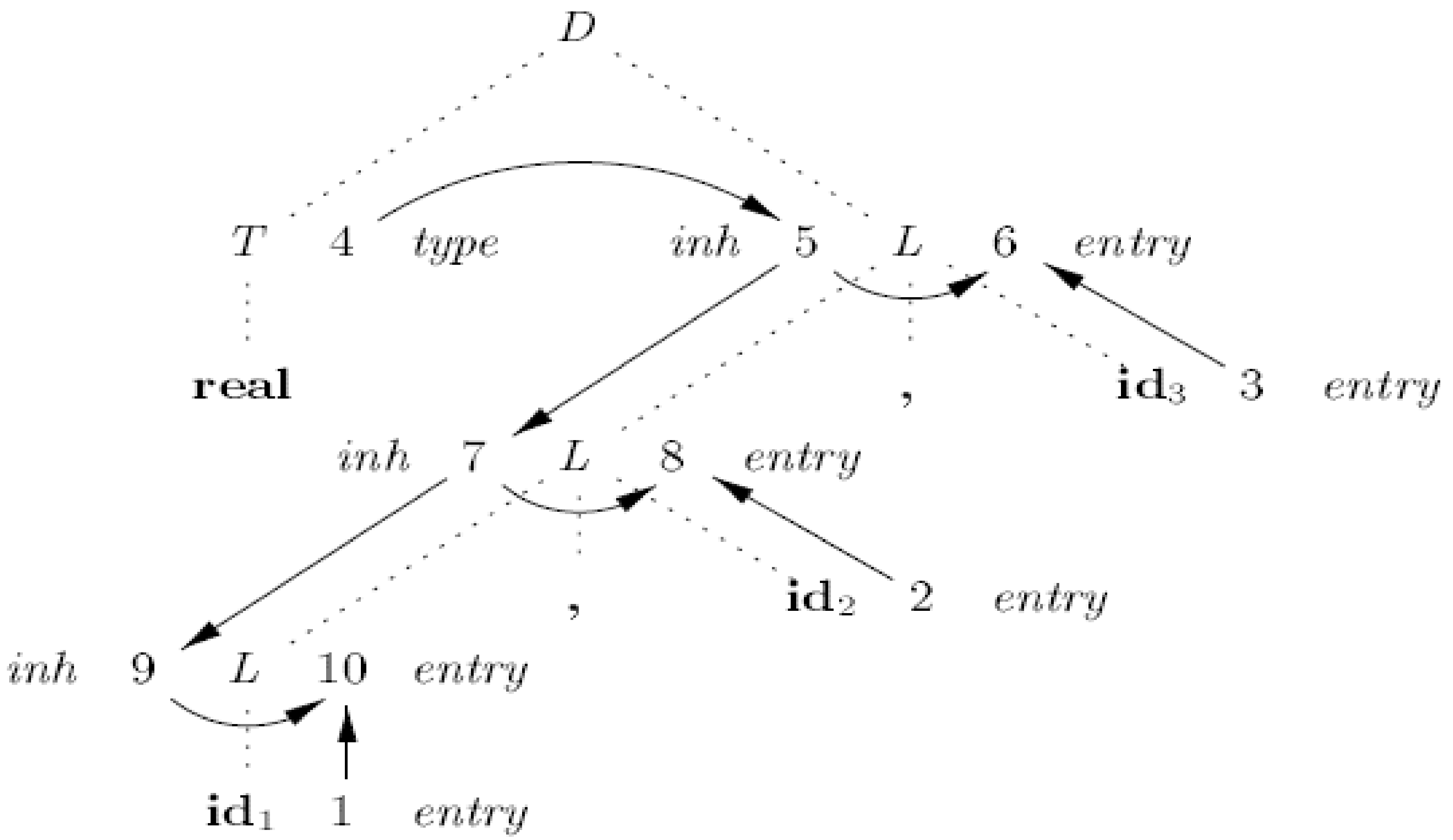
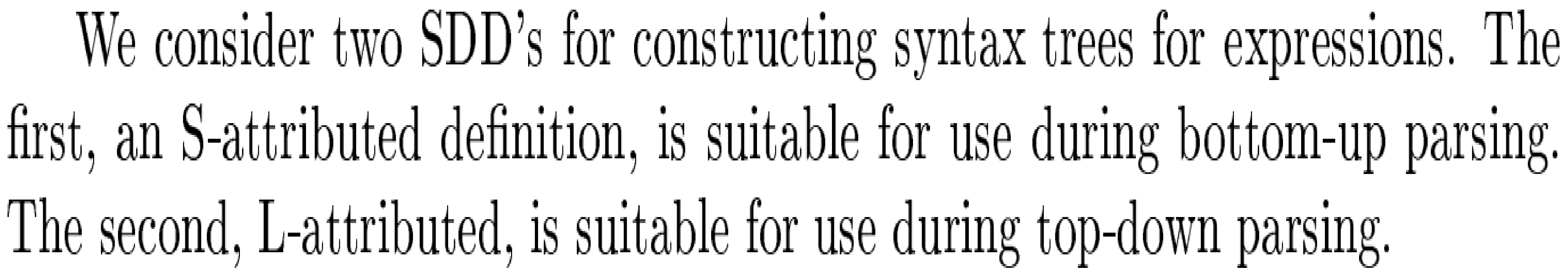


Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Applications of SDT



We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

- Construction of Syntax Trees
- Structure of a Type

Construction of Syntax Trees

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions 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. The objects will have additional fields as follows:

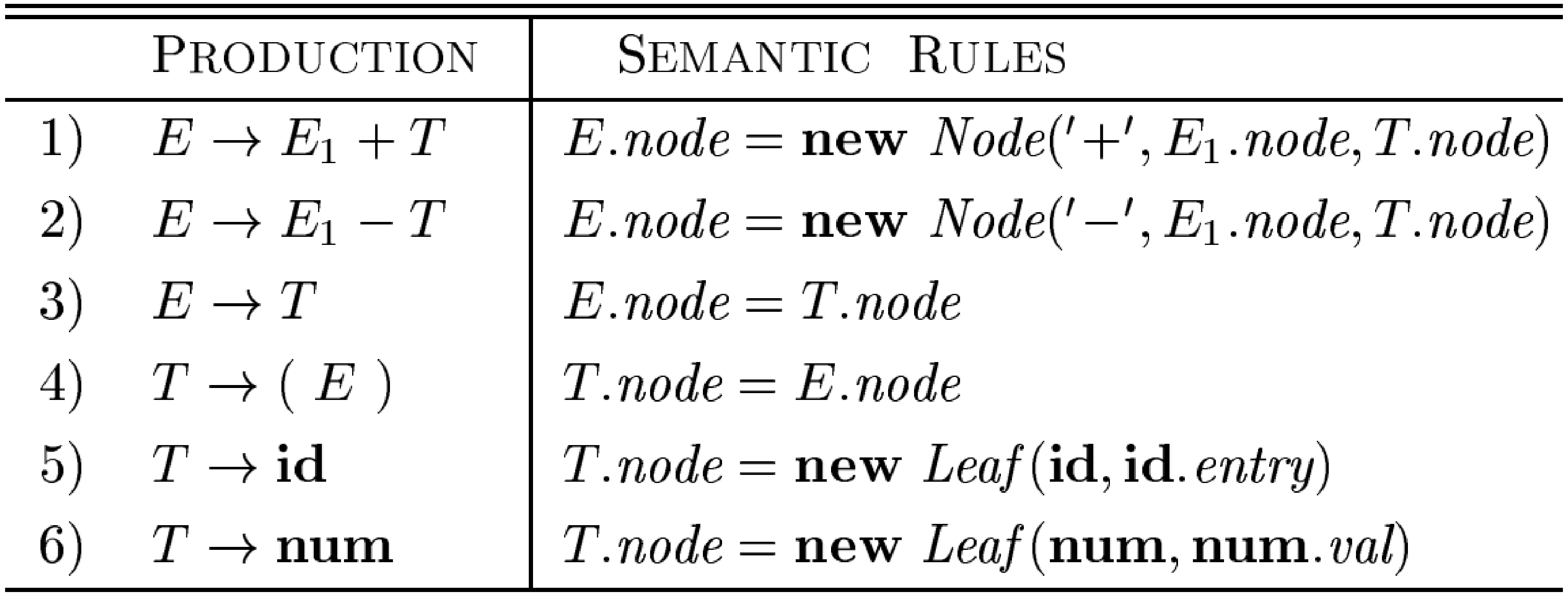
- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf*(*op*, *val*) creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.

- 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: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, \ldots, c_k$.

Construction of Syntax Trees

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.

Construction of Syntax Trees

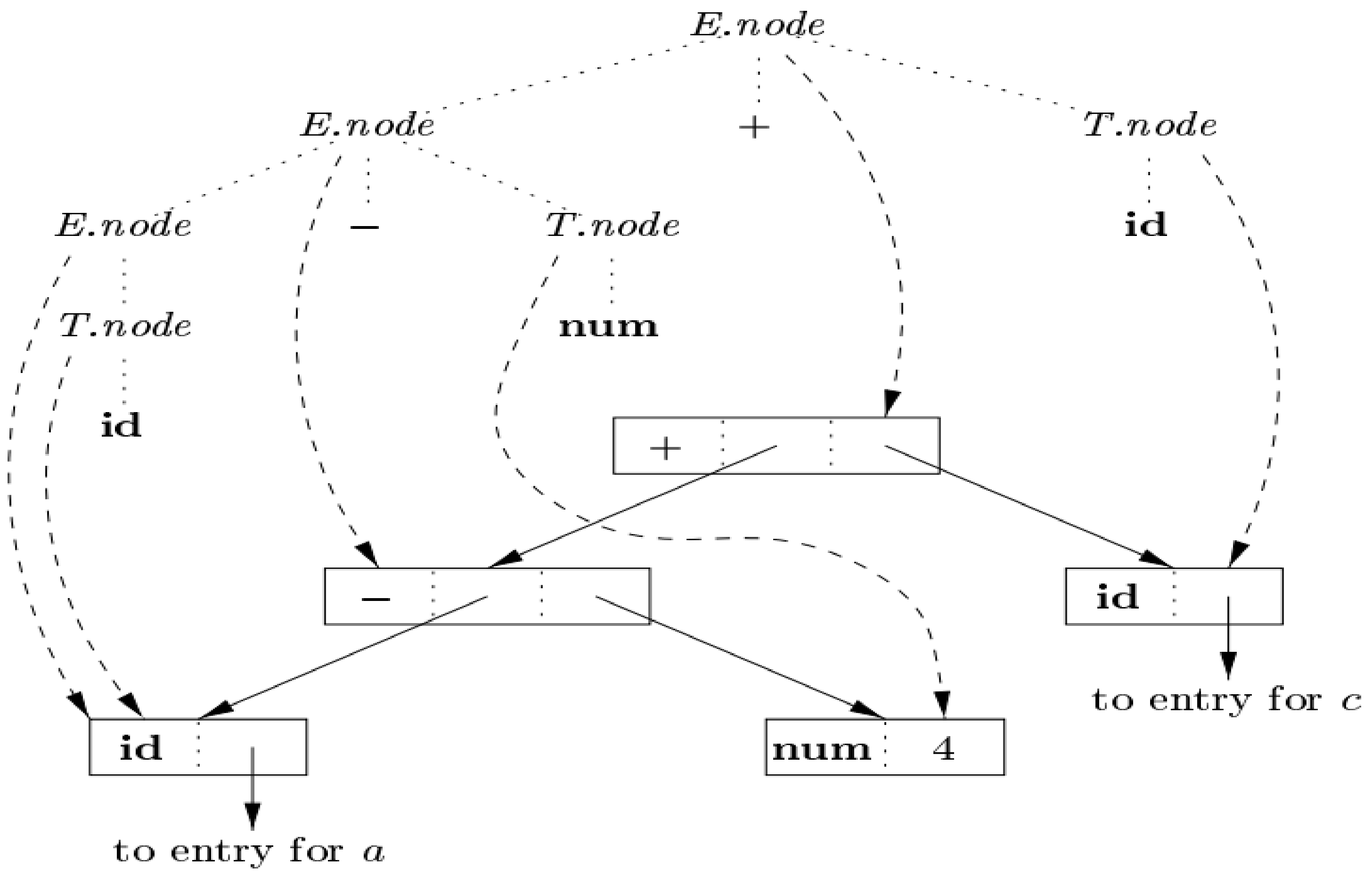


	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$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

Figure 5.10: Constructing syntax trees for simple expressions

Construction of Syntax Trees

Figure 5.11 shows the construction of a syntax tree for the input $a - 4 + c$.



Construction of Syntax Trees

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 shown in Fig. 5.12 ends with p_5 pointing to the root of the constructed syntax tree.

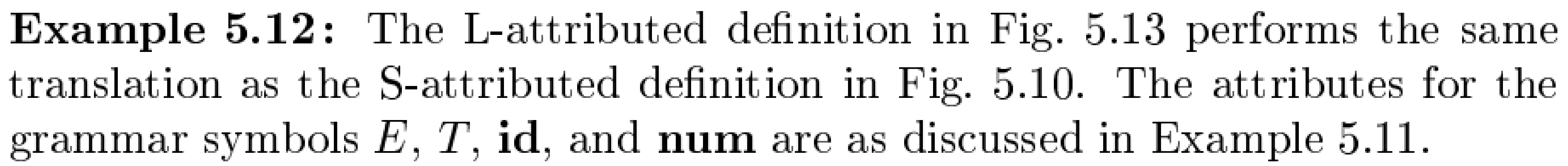
Construction of Syntax Trees

Fig. 5.12 ends with p_5 pointing to the root of the constructed 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)  $p_5 = \textbf{new}\ Node('+', p_3, p_4);$
```

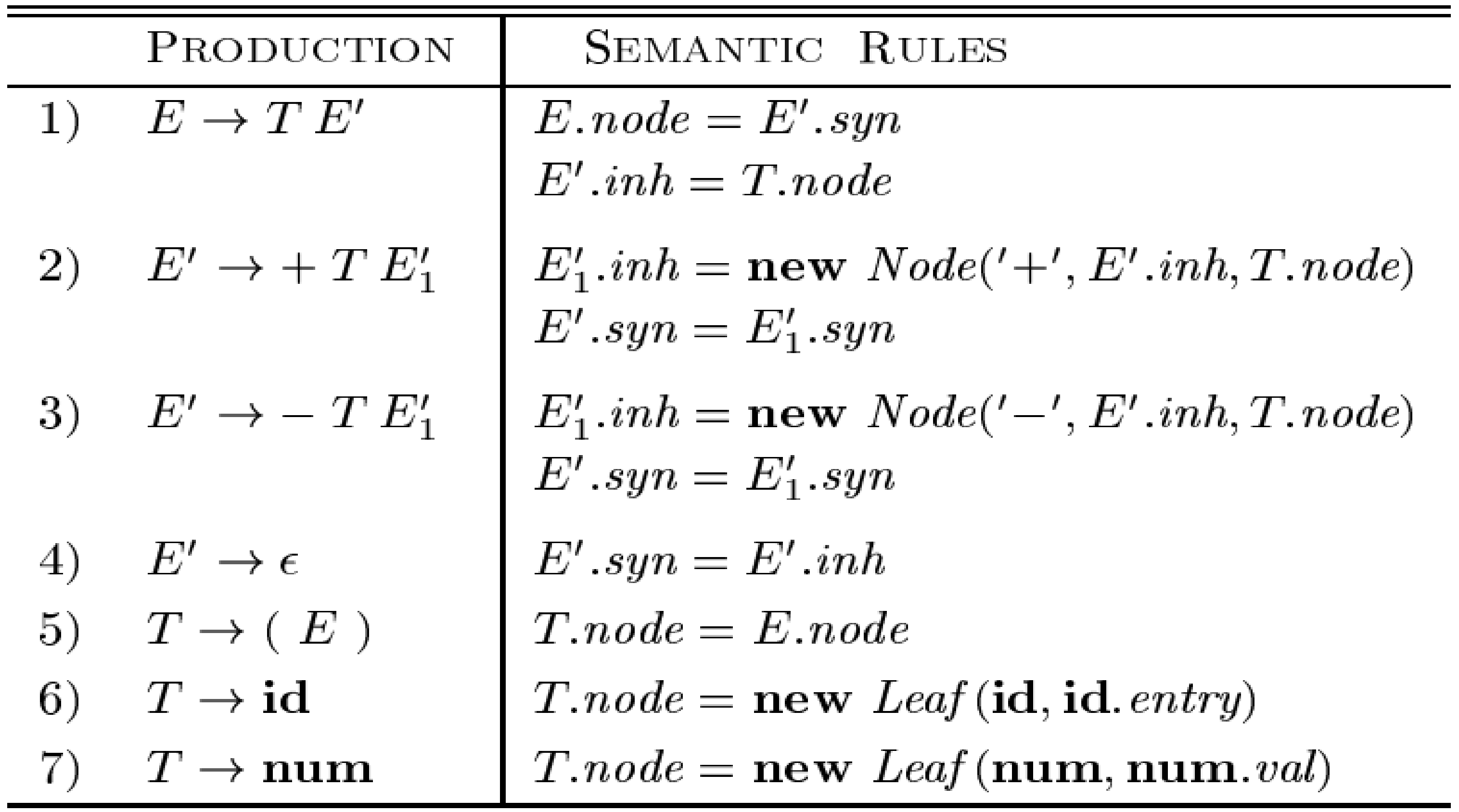
Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

Construction of Syntax Trees



Example 5.12: The L-attributed definition in Fig. 5.13 performs the same translation as the S-attributed definition in Fig. 5.10. The attributes for the grammar symbols E, T, **id**, and **num** are as discussed in Example 5.11.

Construction of Syntax Trees



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

Figure 5.13: Constructing syntax trees during top-down parsing

Construction of Syntax Trees

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5.3. In the desk-calculator example, a term $x * y$ was evaluated by passing x as an inherited attribute, since x and $* y$ appeared in different portions of the parse tree. 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. Nonterminal E' is the counterpart of nonterminal T' in Example 5.3. Compare the dependency graph for $a - 4 + c$ in Fig. 5.14 with that for $3 * 5$ in Fig. 5.7.

Construction of Syntax Trees

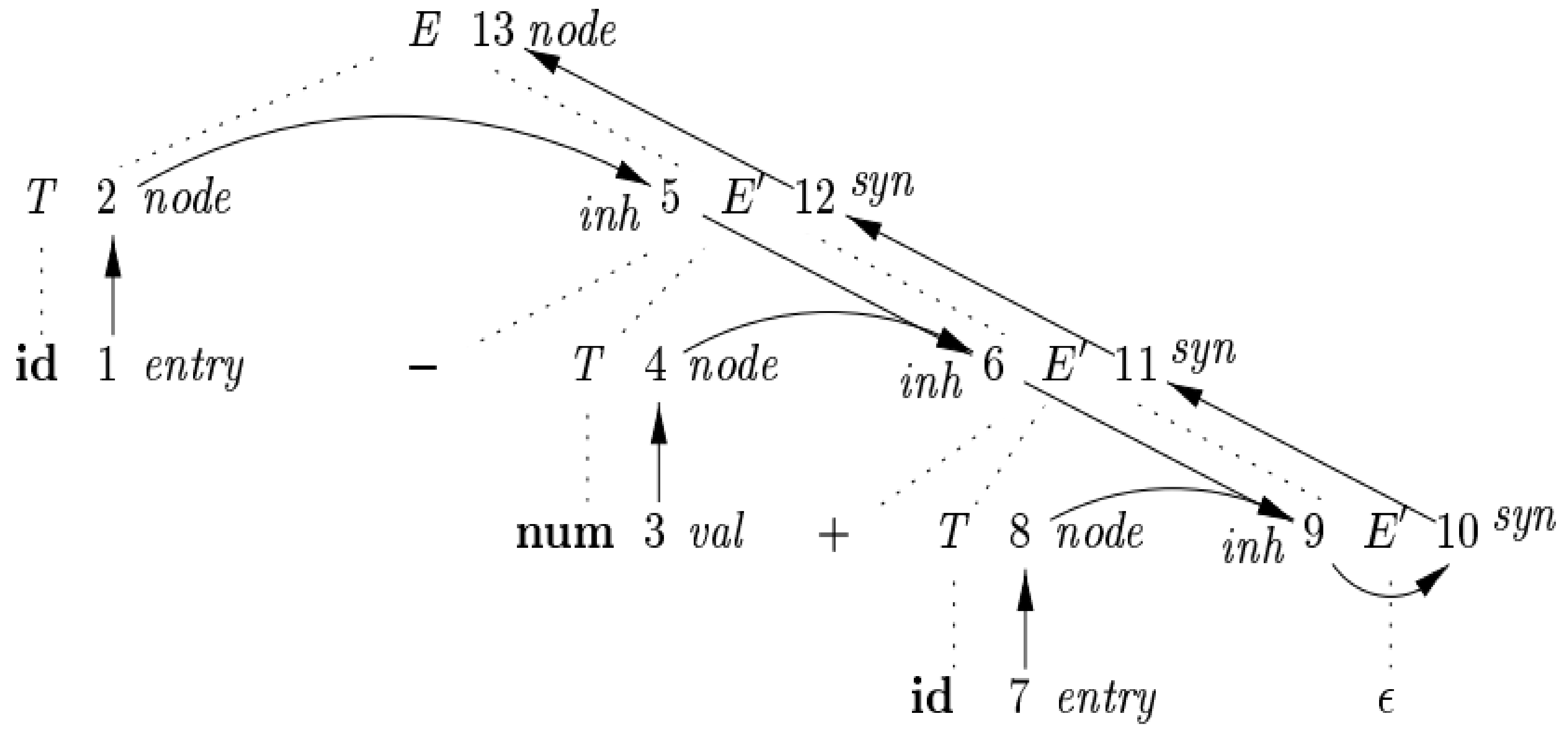
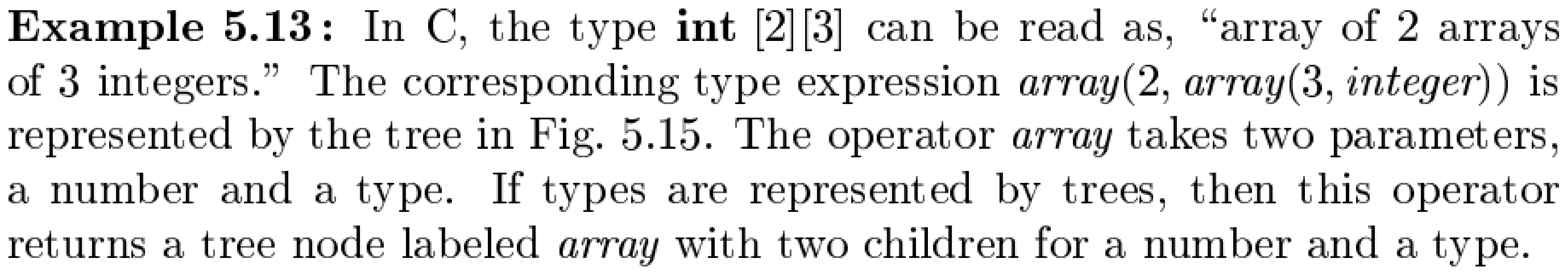


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

Structure of a Type



Example 5.13: In C, the type **int** [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression *array*(2, *array*(3, *integer*)) is represented by the tree in Fig. 5.15. The operator *array* takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled *array* with two children for a number and a type.

Structure of a Type

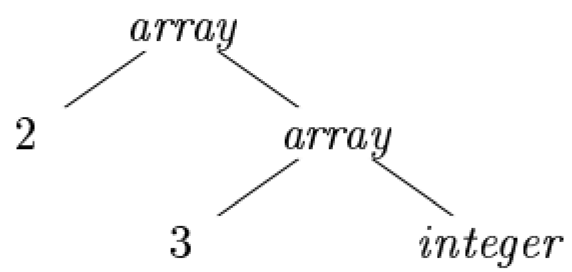
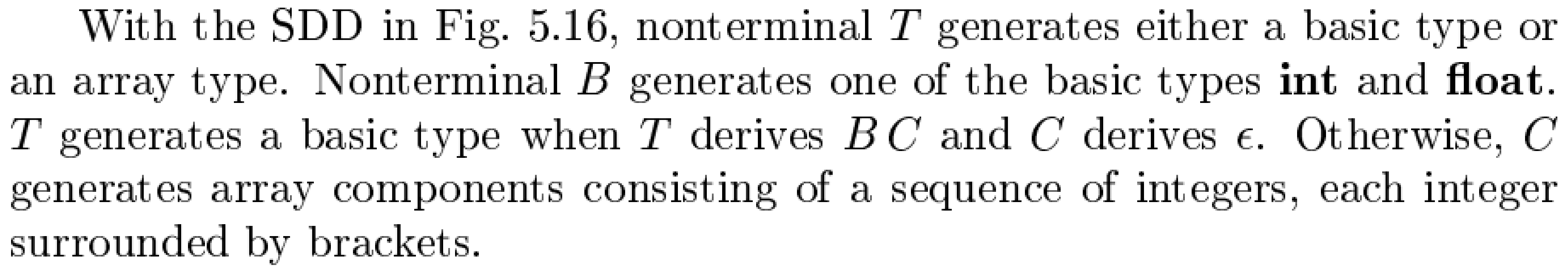


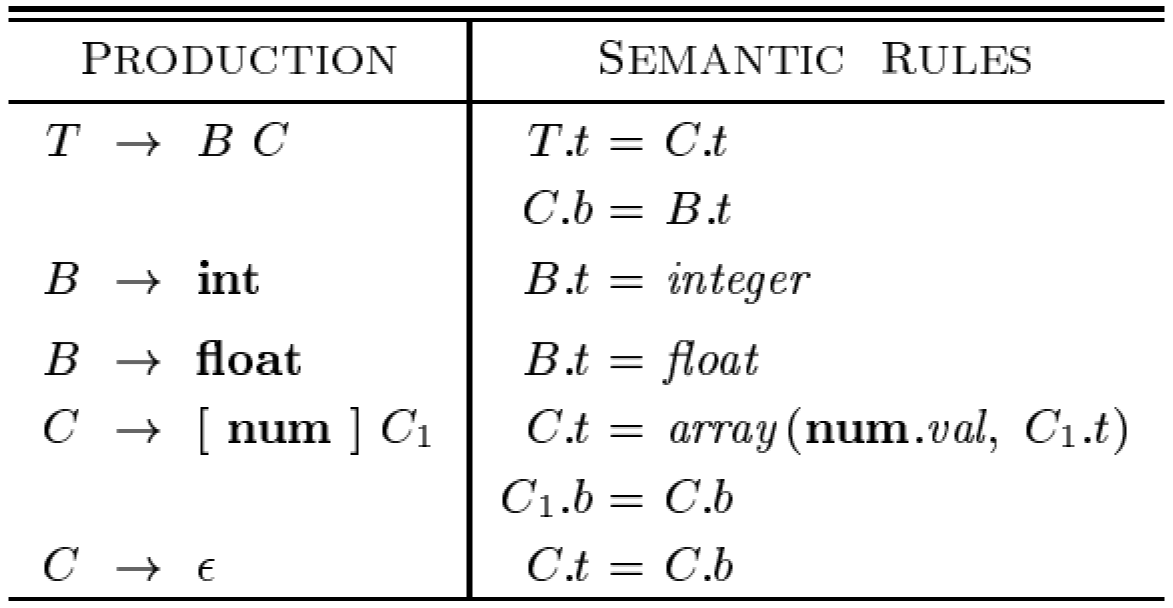
Figure 5.15: Type expression for **int**[2][3]

Structure of a Type



With the SDD in Fig. 5.16, nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types **int** and **float**. T generates a basic type when T derives $B\,C$ and C derives ϵ. Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

Structure of a Type

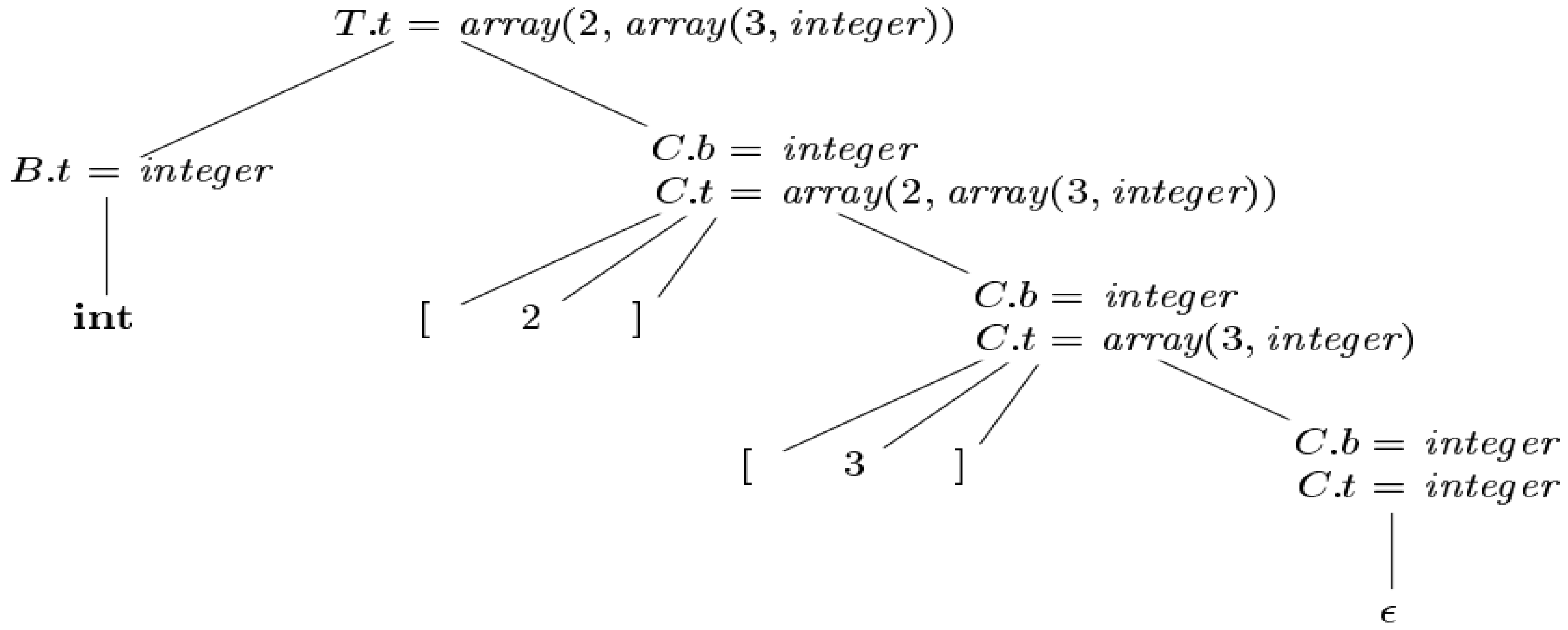


Production	Semantic Rules
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = \mathit{integer}$
$B \rightarrow \textbf{float}$	$B.t = \mathit{float}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = \mathit{array}(\textbf{num}.\mathit{val},\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow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

Figure 5.16: T generates either a basic type or an array type

Structure of a Type

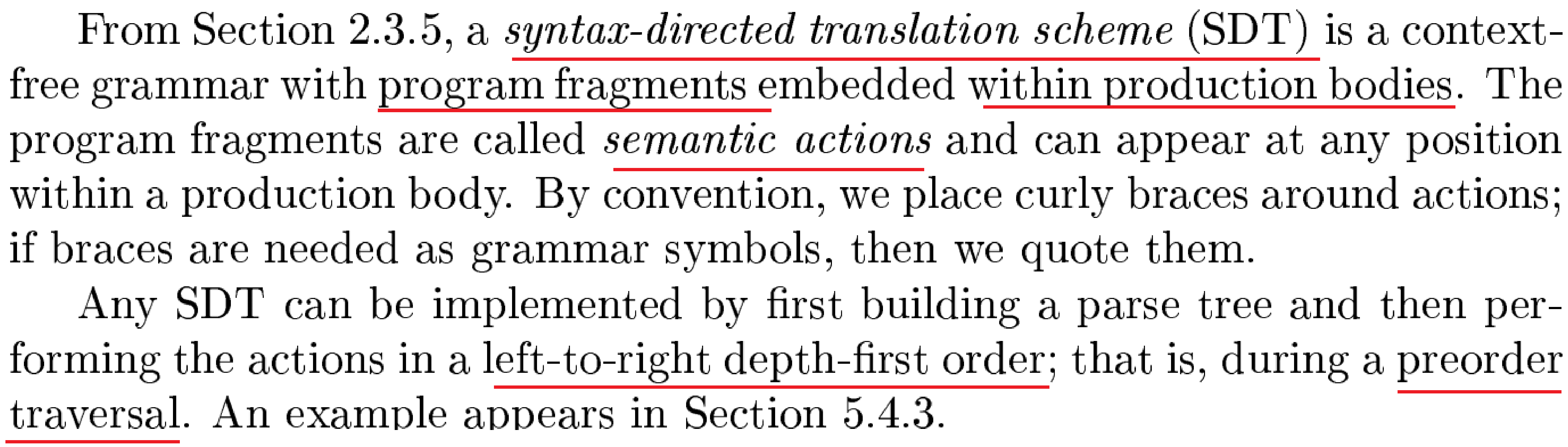
An annotated parse tree for the input string **int** [2] [3] is shown in Fig. 5.17.



$T.t = array(2, array(3, integer))$
$B.t = integer$
int
$C.b = integer$
$C.t = array(2, array(3, integer))$
[
2
]
$C.b = integer$
$C.t = array(3, integer)$
[
3
]
$C.b = integer$
$C.t = integer$
ϵ

Figure 5.17: Syntax-directed translation of array types

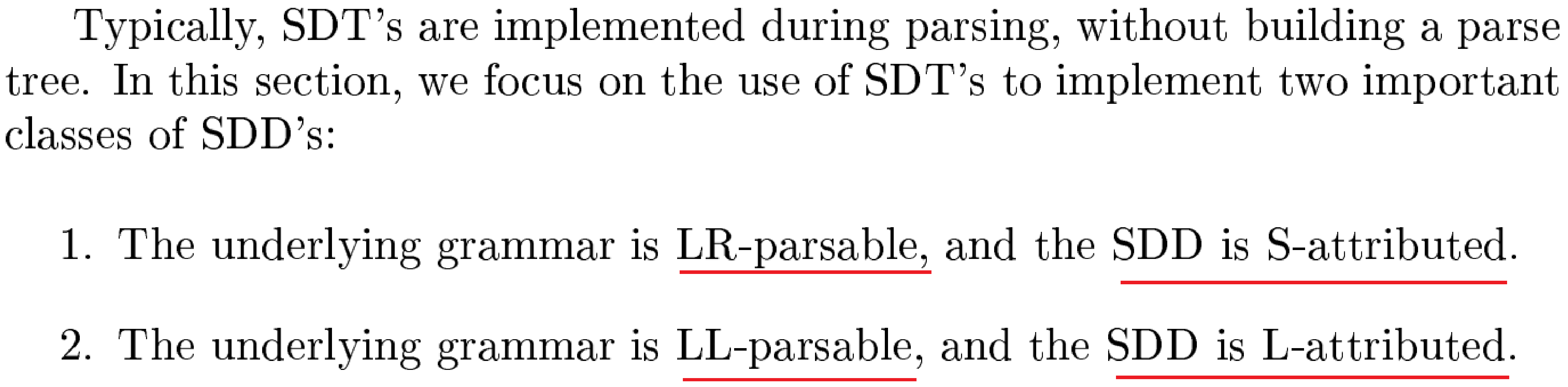
Syntax Directed Translation Schemes



From Section 2.3.5, a *syntax-directed translation scheme* (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and 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. An example appears in Section 5.4.3.

Syntax Directed Translation Schemes



Typically, SDT's are implemented during parsing, without building a parse tree. In this section, we focus on the use of SDT's to implement two important classes of SDD's:

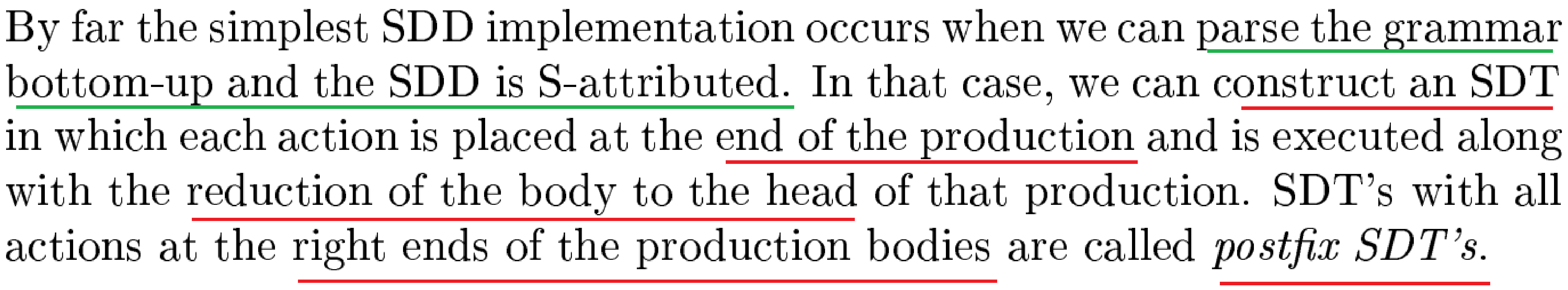
1. The underlying grammar is LR-parsable, and the SDD is S-attributed.

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

Syntax Directed Translation Schemes

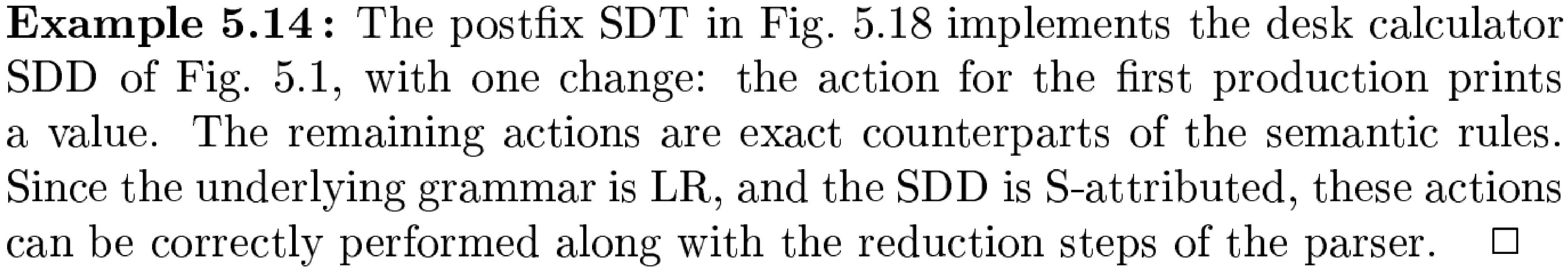
- Postfix Translation Schemes
- Parser-stack Implementation of Postfix SDT's
- SDT's with Action Inside Productions
- Eliminating Left Recursion from SDT's

Postfix Translation Schemes



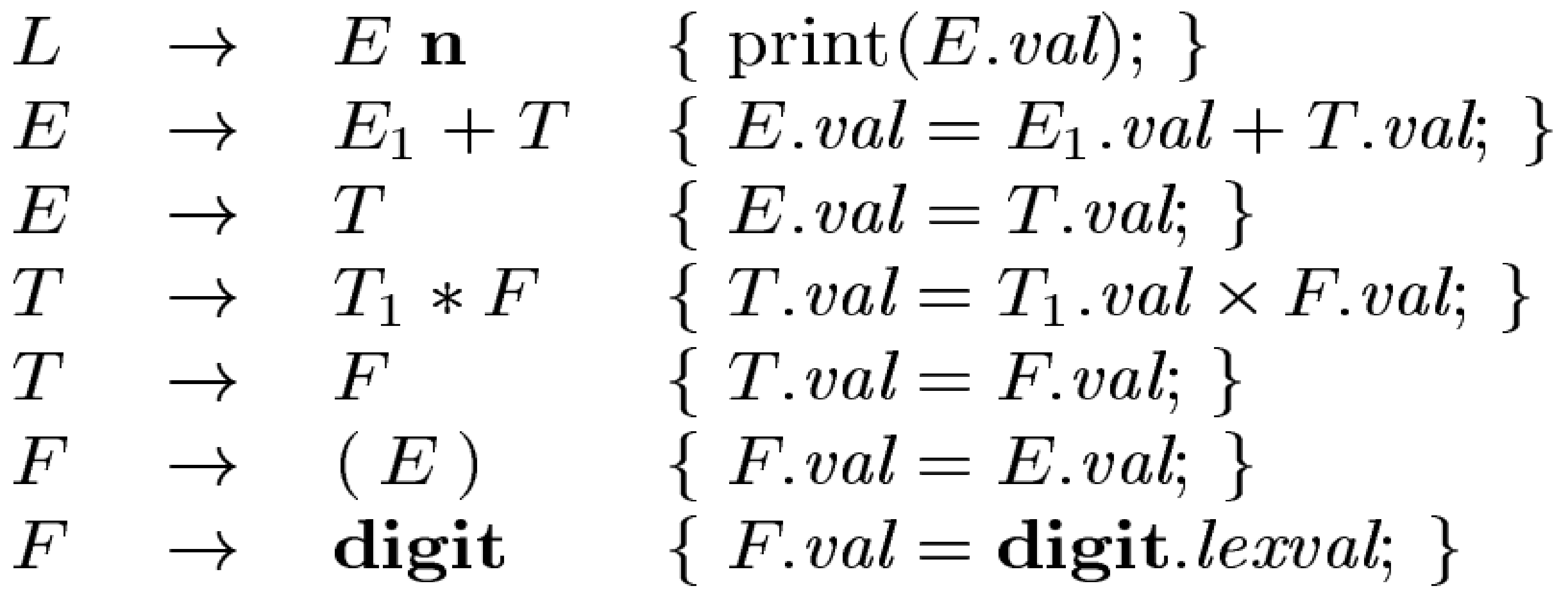
By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

Postfix Translation Schemes



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. □

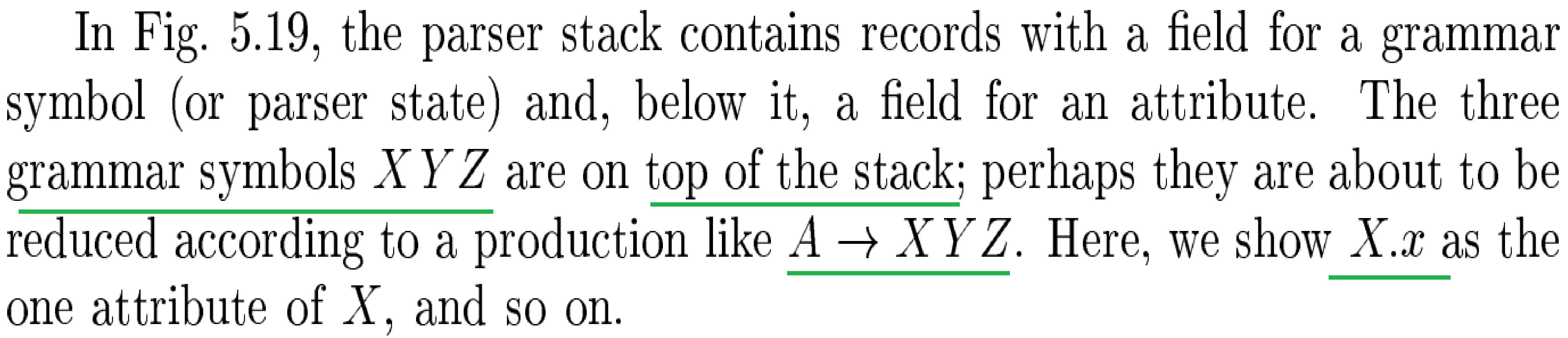
Postfix Translation Schemes



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	$\rightarrow$	$\mathbf{digit}$	$\{\ F.val = \mathbf{digit}.lexval;\ \}$

Figure 5.18: Postfix SDT implementing the desk calculator

Parser-stack Implementation of Postfix SDT's



In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols XYZ are on top of the stack; perhaps they are about to be reduced according to a production like $A \rightarrow XYZ$. Here, we show $X.x$ as the one attribute of X, and so on.

Parser-stack Implementation of Postfix SDT’s

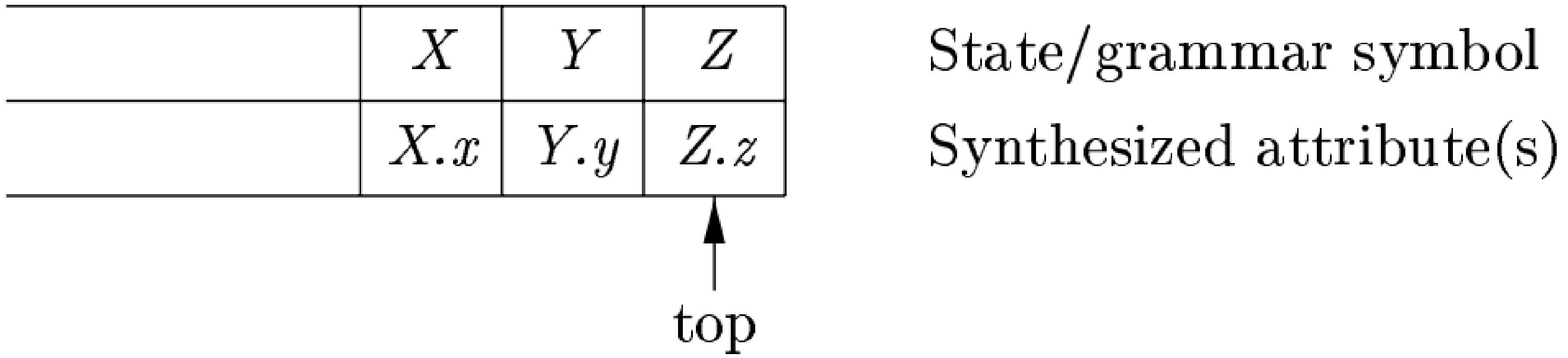
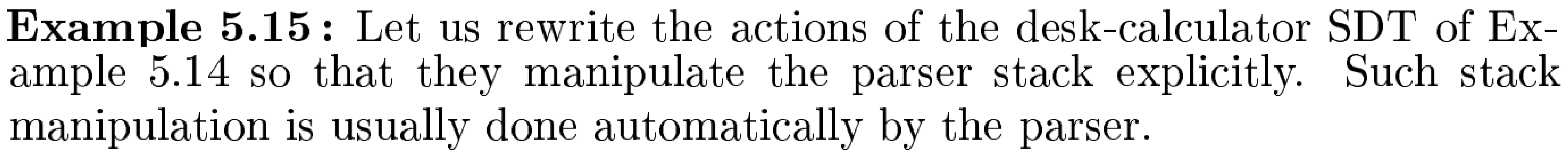


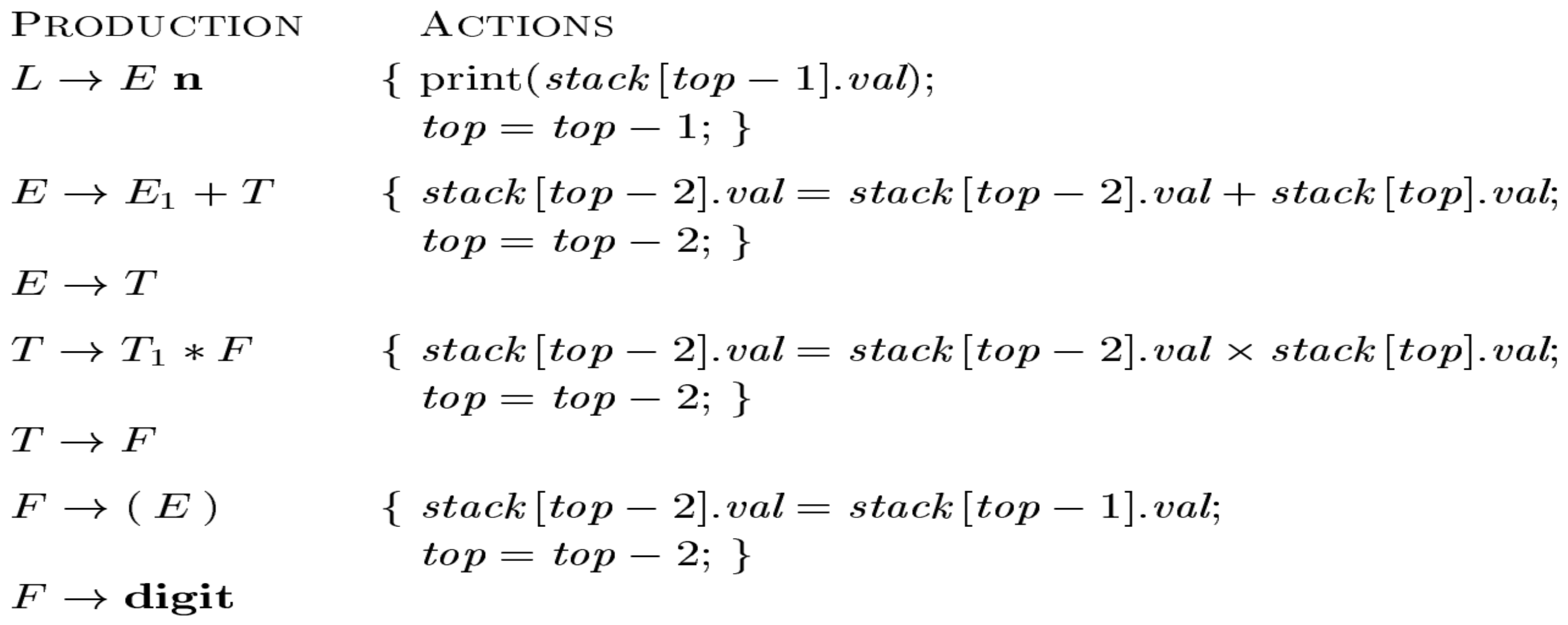
Figure 5.19: Parser stack with a field for synthesized attributes

Parser-stack Implementation of Postfix SDT's



Example 5.15: Let us rewrite the actions of the desk-calculator SDT of Example 5.14 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

Parser-stack Implementation of Postfix SDT's



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

Figure 5.20: Implementing the desk calculator on a bottom-up parsing stack

SDT's with Action Inside Productions

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 $B \to X \; \{a\} \; Y$, 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 nonterminal). 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 nonterminal) or check for Y on the input (if Y is a terminal).

SDT's with Action Inside Productions

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.

SDT’s with Action Inside Productions

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

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

SDT's with Action Inside Productions

For instance, Fig. 5.22 shows the parse tree for expression $3 * 5 + 4$ with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: $+ * 3\ 5\ 4$.

SDT's with Action Inside Productions

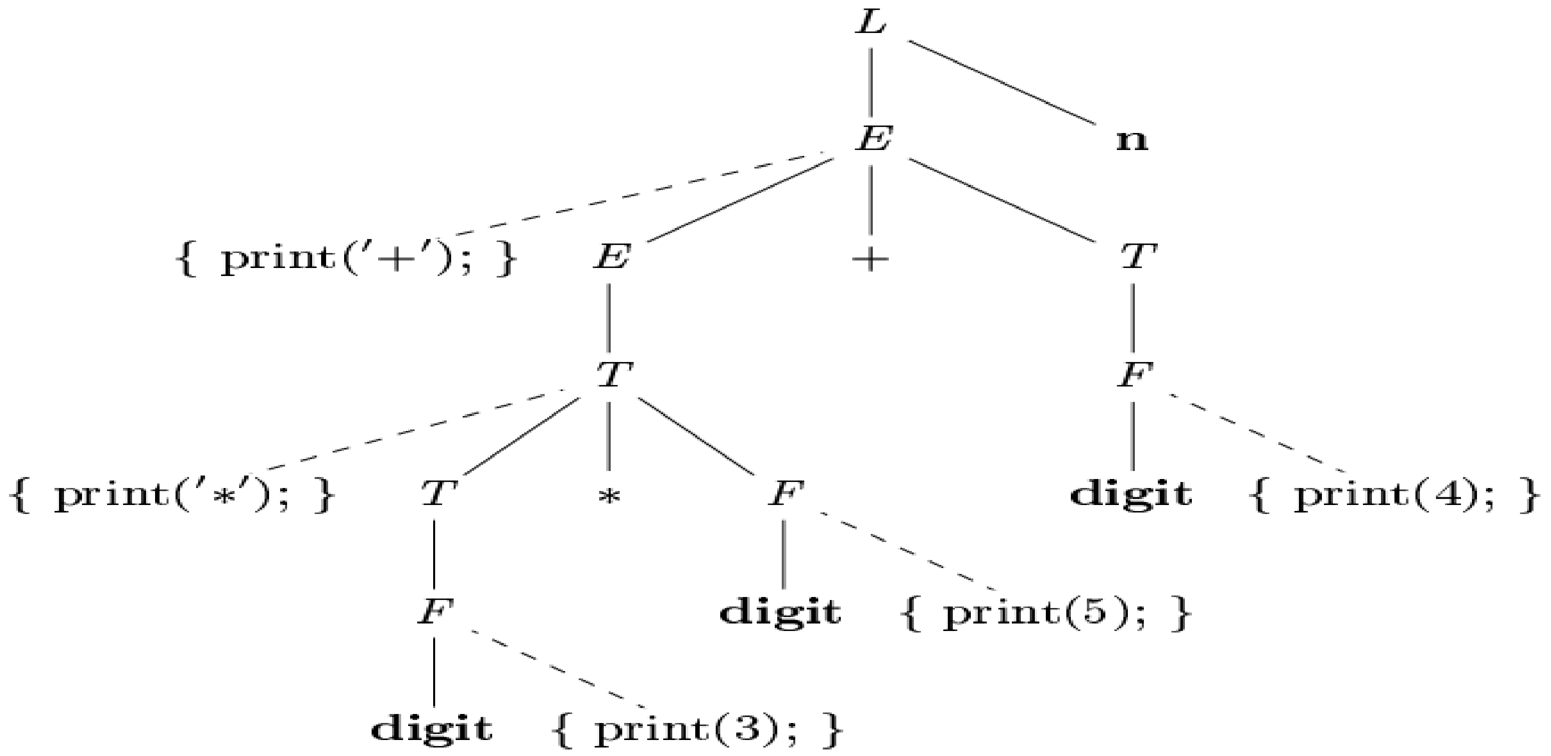


Figure 5.22: Parse tree with actions embedded

Eliminating Left Recursion from SDT’s

When transforming the grammar, treat the actions as if they were terminal symbols.

The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

Eliminating Left Recursion from SDT's

The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal R (for "remainder") of the first production:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

If β does not begin with A, then A no longer has a left-recursive production. In regular-definition terms, with both sets of productions, A is defined by $\beta(\alpha)^*$.

Eliminating Left Recursion from SDT's

Example 5.17: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$\begin{array}{lcll} E & \rightarrow & E_1 + T & \{\ \text{print}('+');\ \} \\ E & \rightarrow & T & \end{array}$$

Eliminating Left Recursion from SDT's

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \;=\; + \; T \; \{ \; \text{print}('+'); \; \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$\begin{array}{lcl} E & \rightarrow & T \; R \\ R & \rightarrow & + \; T \; \{ \; \text{print}('+'); \; \} \; R \\ R & \rightarrow & \epsilon \end{array}$$

Eliminating Left Recursion from SDT's

We shall give a general schema for the case of a single recursive production, a single nonrecursive production, and a single attribute of the left-recursive nonterminal; the generalization to many productions of each type is not hard, but is notationally cumbersome. Suppose that the two productions are

Eliminating Left Recursion from SDT’s

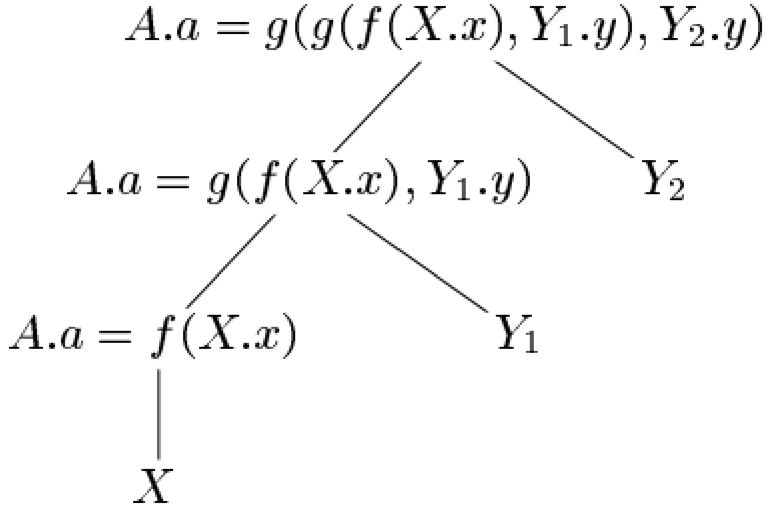
$$\begin{array}{lcl} A & \rightarrow & A_1 \, Y \, \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X \, \{A.a = f(X.x)\} \end{array}$$

Eliminating Left Recursion from SDT's

$$\begin{array}{lcl} A & \rightarrow & X\, R \\ R & \rightarrow & Y\, R \mid \epsilon \end{array}$$

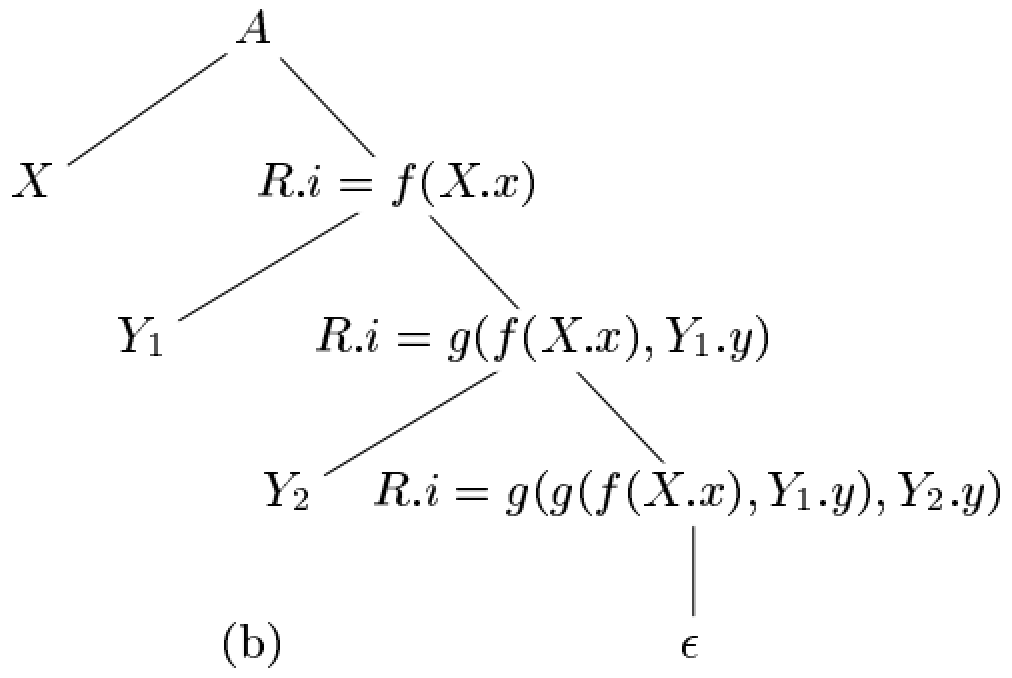
Eliminating Left Recursion from SDT's

$$\begin{array}{lll} A & \rightarrow & A_1 \; Y \; \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X \; \{A.a = f(X.x)\} \end{array}$$



Eliminating Left Recursion from SDT's

$$\begin{aligned} A &\rightarrow X\,R \\ R &\rightarrow Y\,R \mid \epsilon \end{aligned}$$



(b)

Eliminating Left Recursion from SDT's

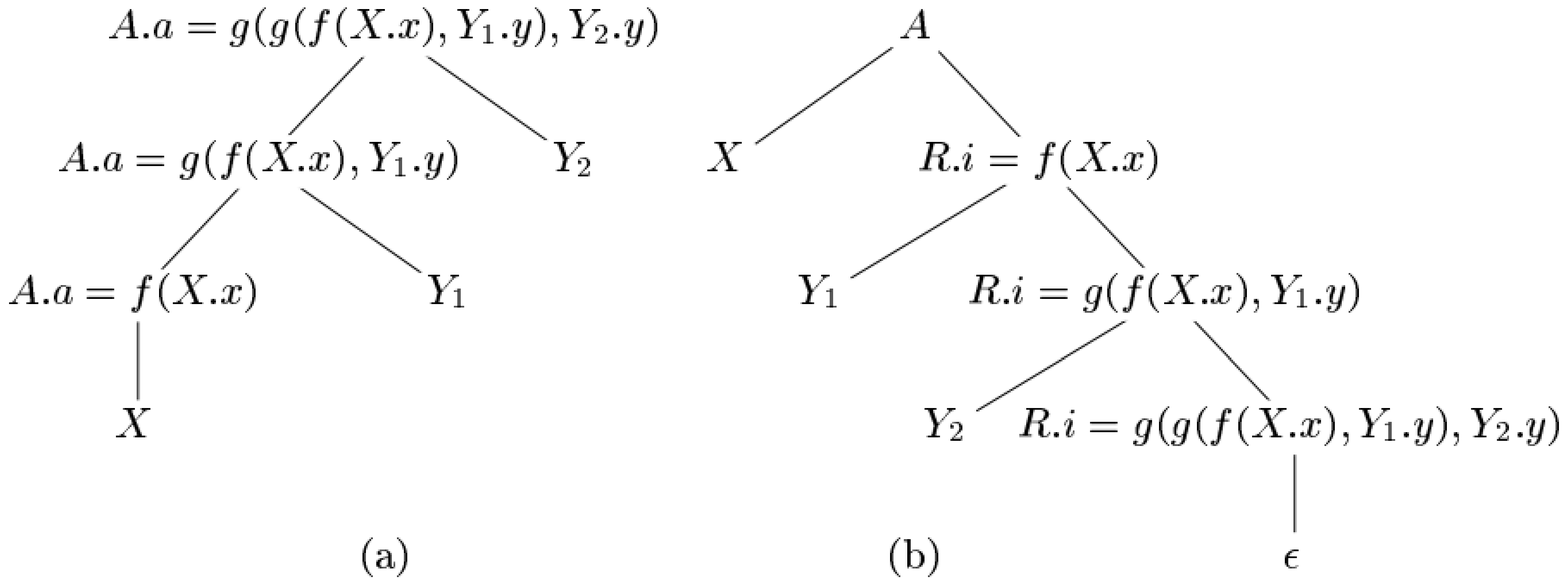


Figure 5.23: Eliminating left recursion from a postfix SDT

Eliminating Left Recursion from SDT’s

$$\begin{array}{lcl} A & \rightarrow & X \;\; \{R.i = f(X.x)\} \;\; R \;\; \{A.a = R.s\} \\ R & \rightarrow & Y \;\; \{R_1.i = g(R.i, Y.y)\} \;\; R_1 \;\; \{R.s = R_1.s\} \\ R & \rightarrow & \epsilon \;\; \{R.s = R.i\} \end{array}$$

Thank You