Chapter-05

Syntax-Directed Translation

Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a
  - context-free grammar 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
- Attributes may be of any kind: numbers, types, table references, or values, for instance

Types of Attributes for Nonterminals

- We shall deal with two kinds of attributes for nonterminals:
  - Synthesized Attribute
  - Inherited Attribute

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

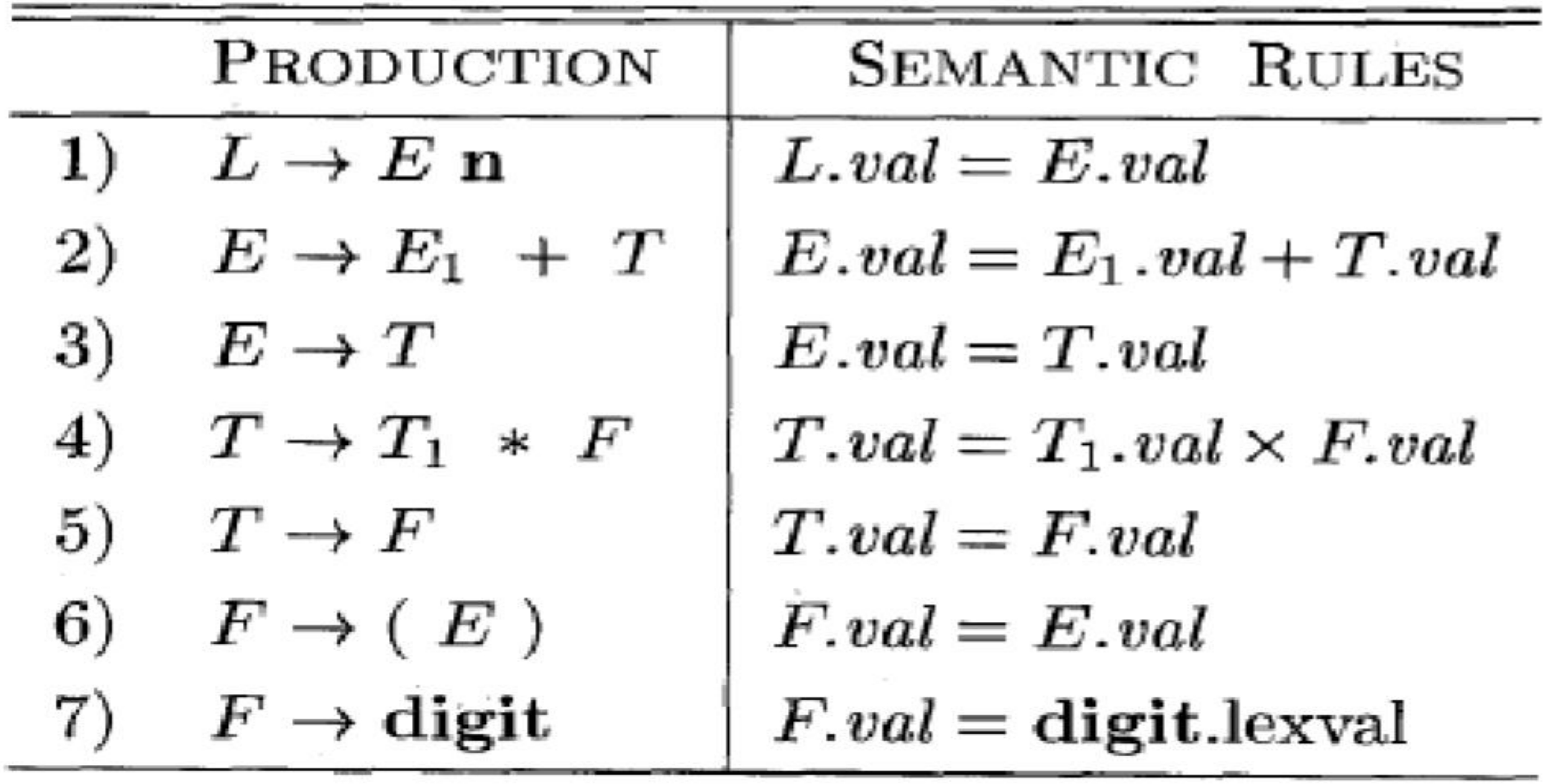
Inherited Attribute

- 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

Attributes for Terminals

- Terminals can have synthesized attributes, but not inherited attributes
- 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

Example 5.1



	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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax-directed definition of a simple desk calculator

S-attributed SDD

- An SDD that involves only synthesized attributes is called S-attributed SDD
- The SDD in the **example 5.1** has this property
- 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

Attribute Grammar

- Simple SDD’s have semantic rules without side effects
- An SDD without side effects is sometimes called an attribute grammar
- The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants

Evaluating an SDD at the Nodes of a Parse Tree

- To visualize the translation specified by an SDD, it helps to work with parse trees
- By constructing an **annotated parse tree** we can evaluate an SDD at the nodes of a parse tree

Annotated Parse Tree

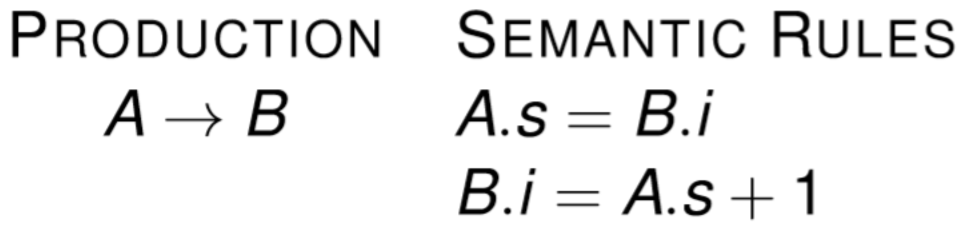
- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree
- **How do we construct an annotated parse tree?**
  - The rules of an SDD are applied by first constructing a parse tree
  - Then using the rules to evaluate all of the attributes at each of the nodes of the parse tree

Annotated Parse Tree(Cont…)

- **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 attributes at all of the children of a node before we can evaluate the 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

Circular Dependency

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

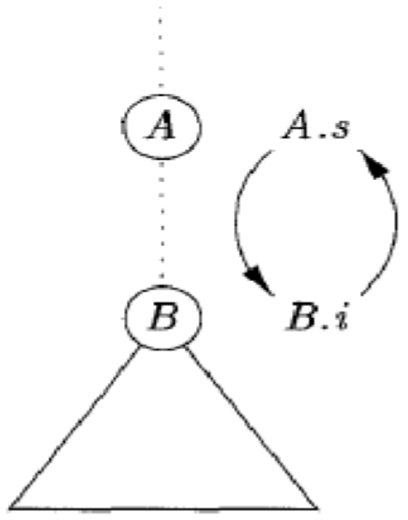


PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

- These rules are circular

Circular Dependency(Cont…)

- 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 in the figure



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

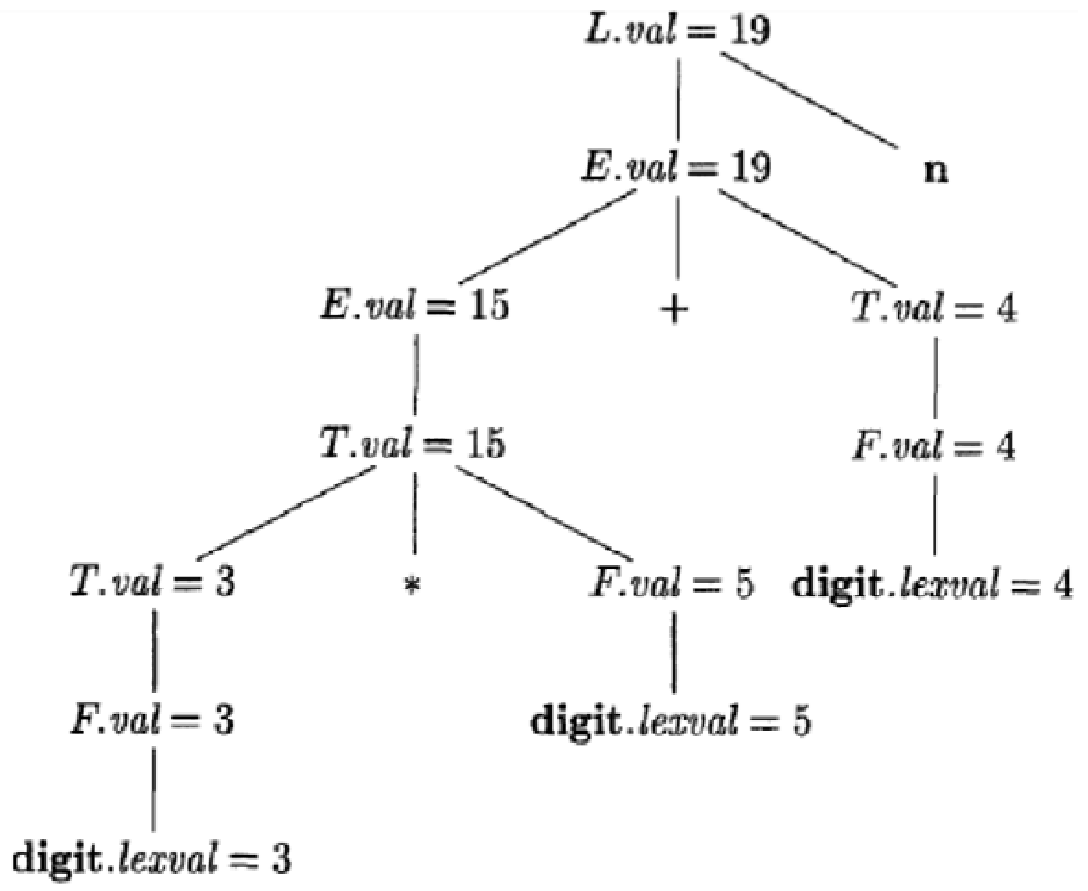
Circular Dependency(Cont…)

- It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate
- Fortunately, there are useful subclasses of SDD’s that are sufficient to guarantee that an order of evaluation exists.

Example 5.2

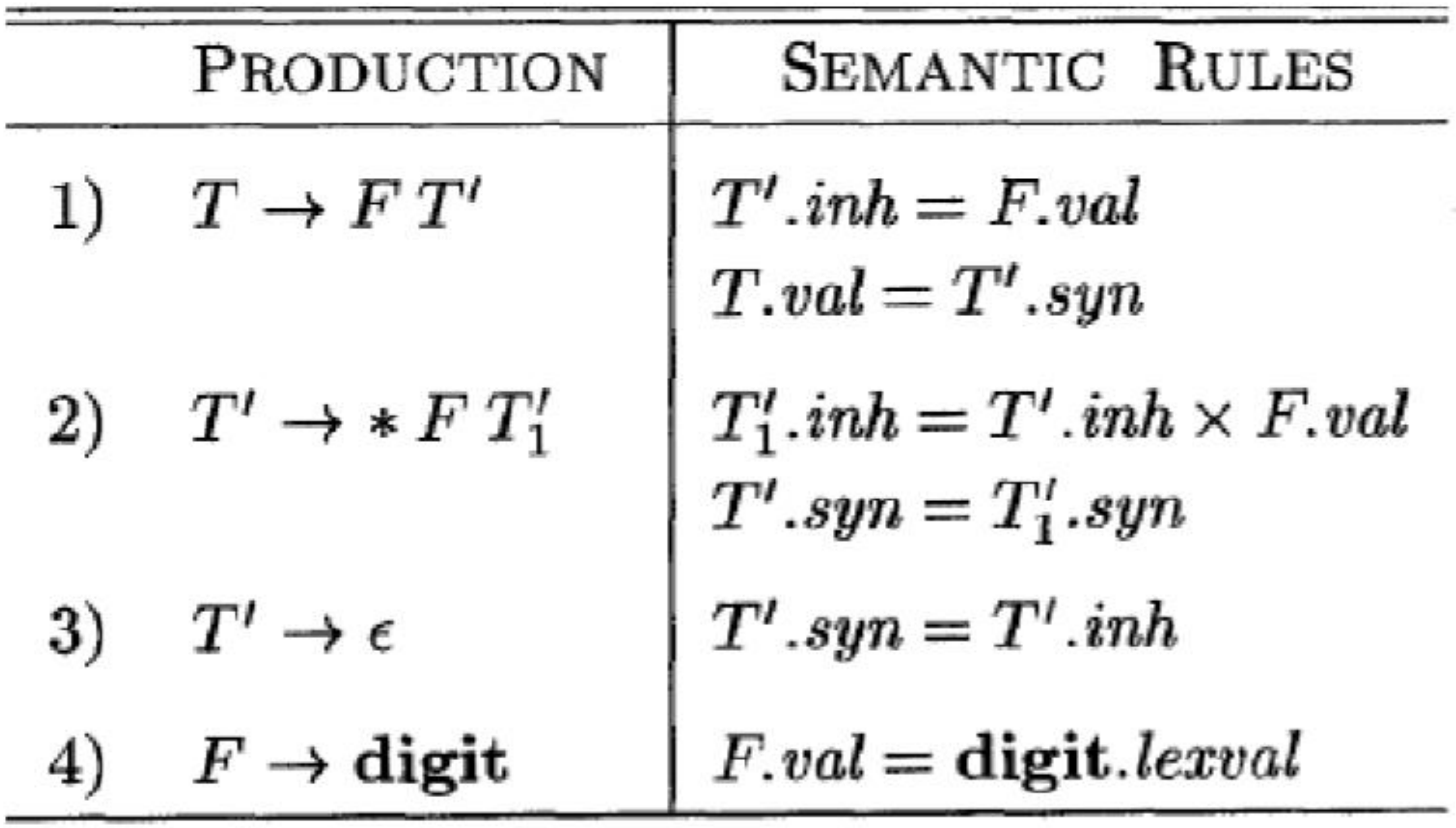
	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax-directed definition of a simple desk calculator



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

Example 5.3



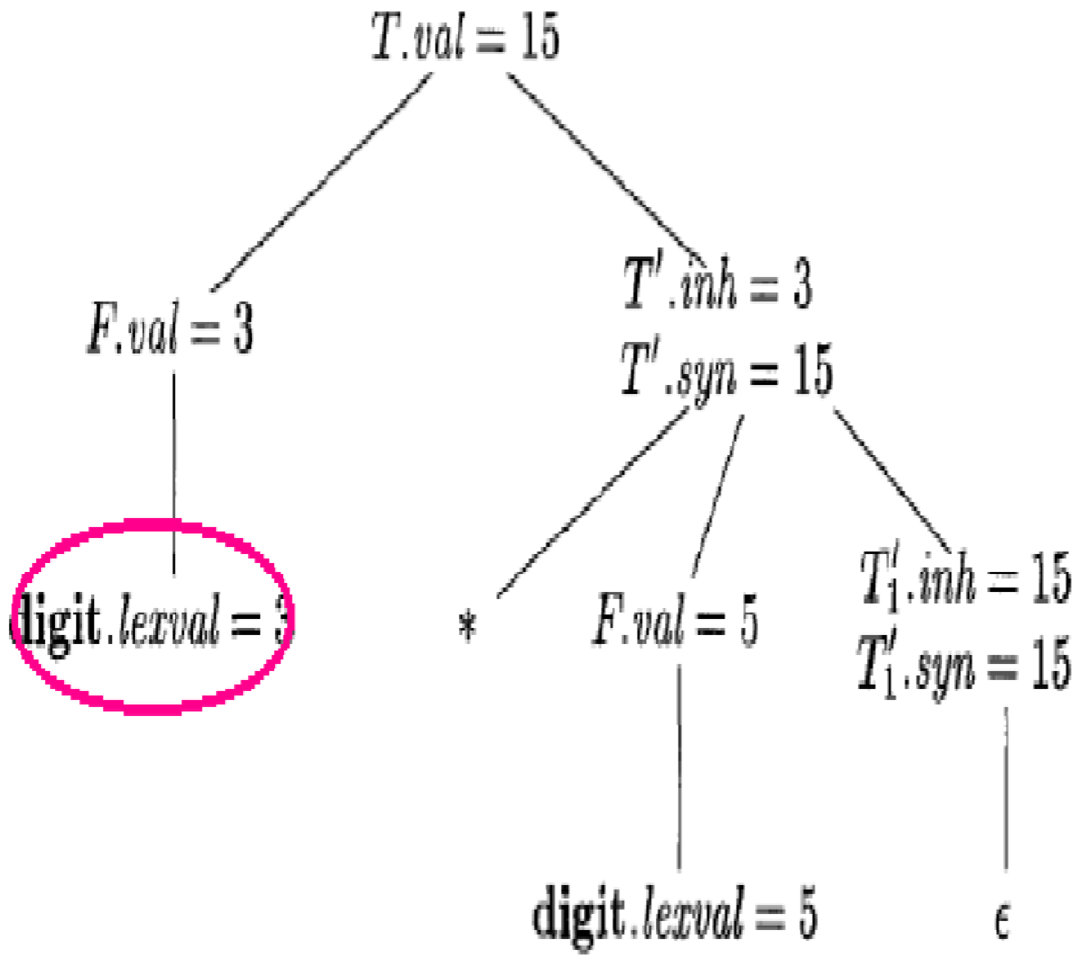
	Production	Semantic Rules
1)	$T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\, F\, T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

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

Example 5.3(Cont…)

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

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

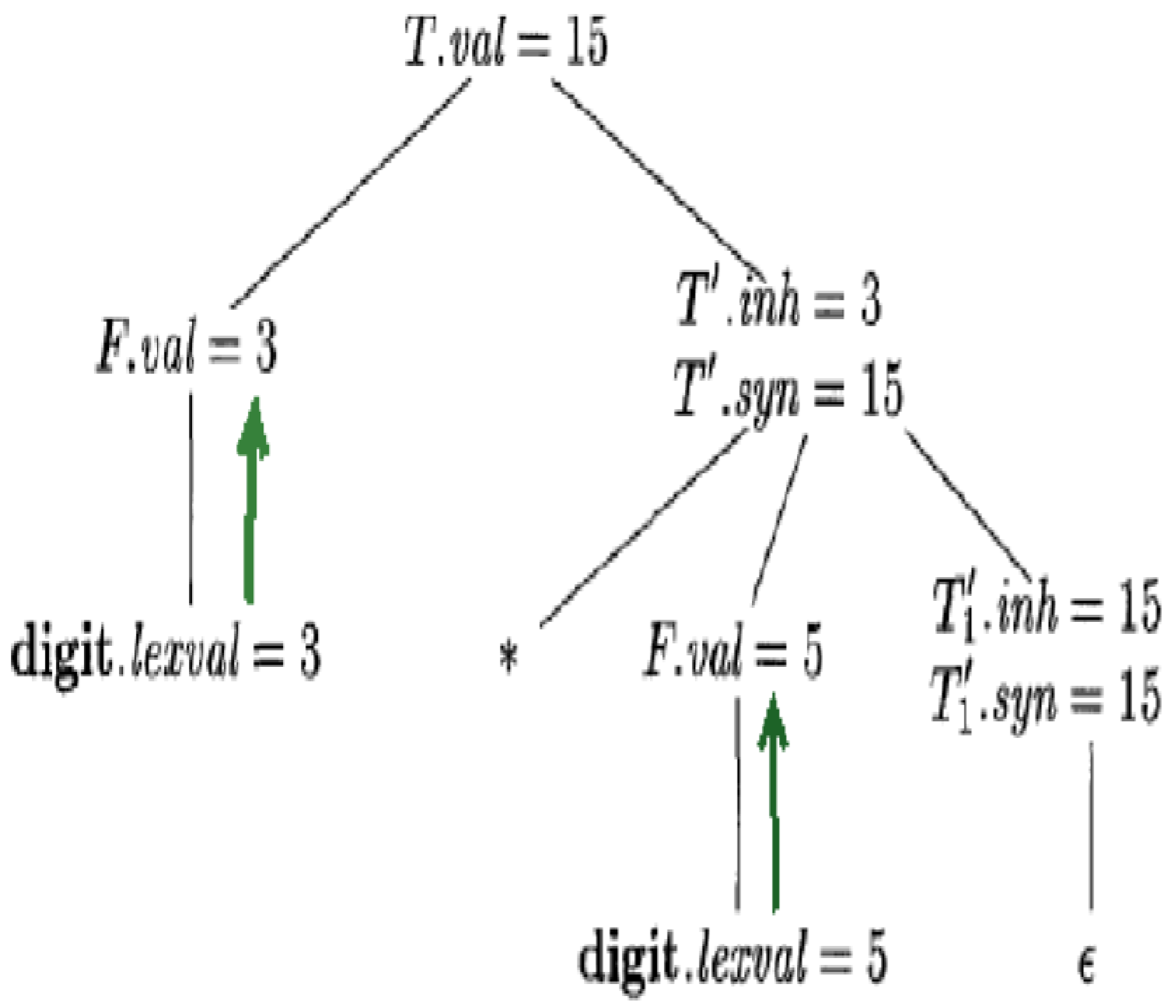


Annotated parse tree for $3 * 5$

Example 5.3(Cont…)

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

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

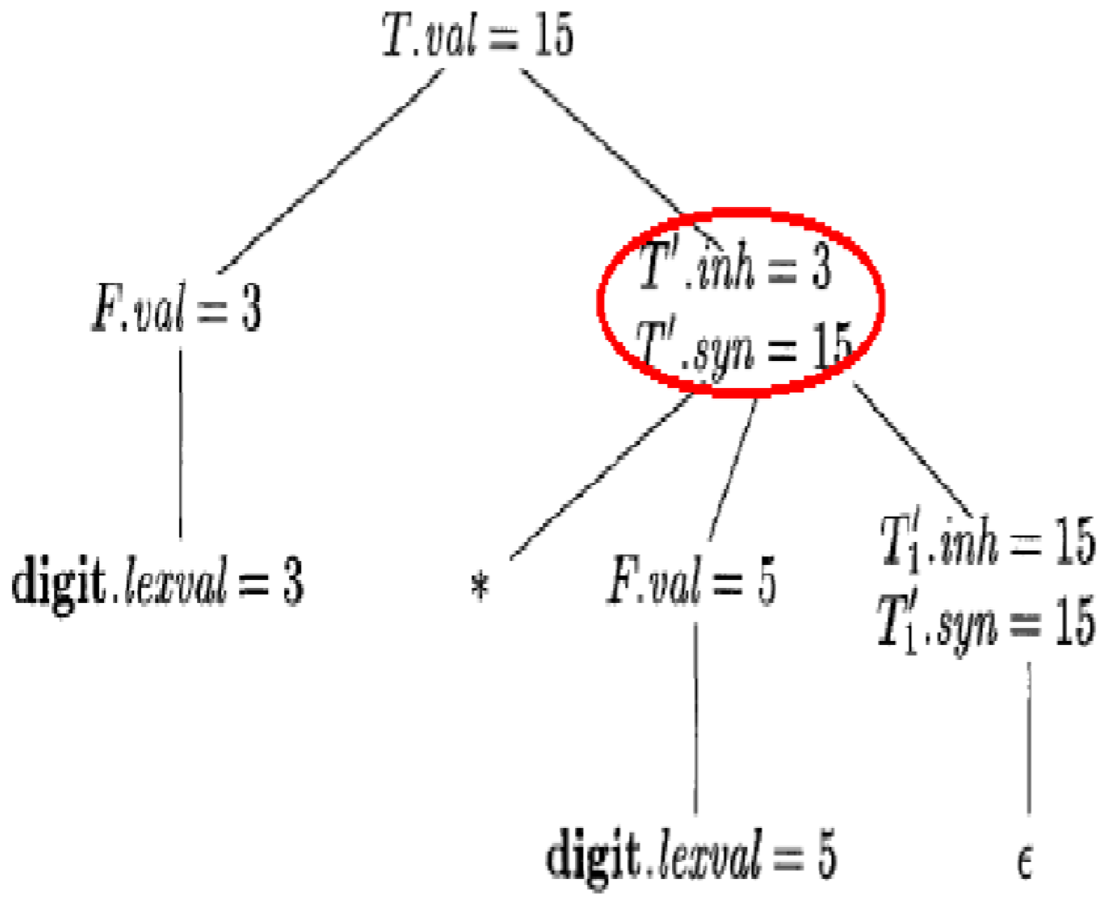


Annotated parse tree for $3 * 5$

Example 5.3(Cont…)

	PRODUCTION	SEMANTIC RULES
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\,F\,T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

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

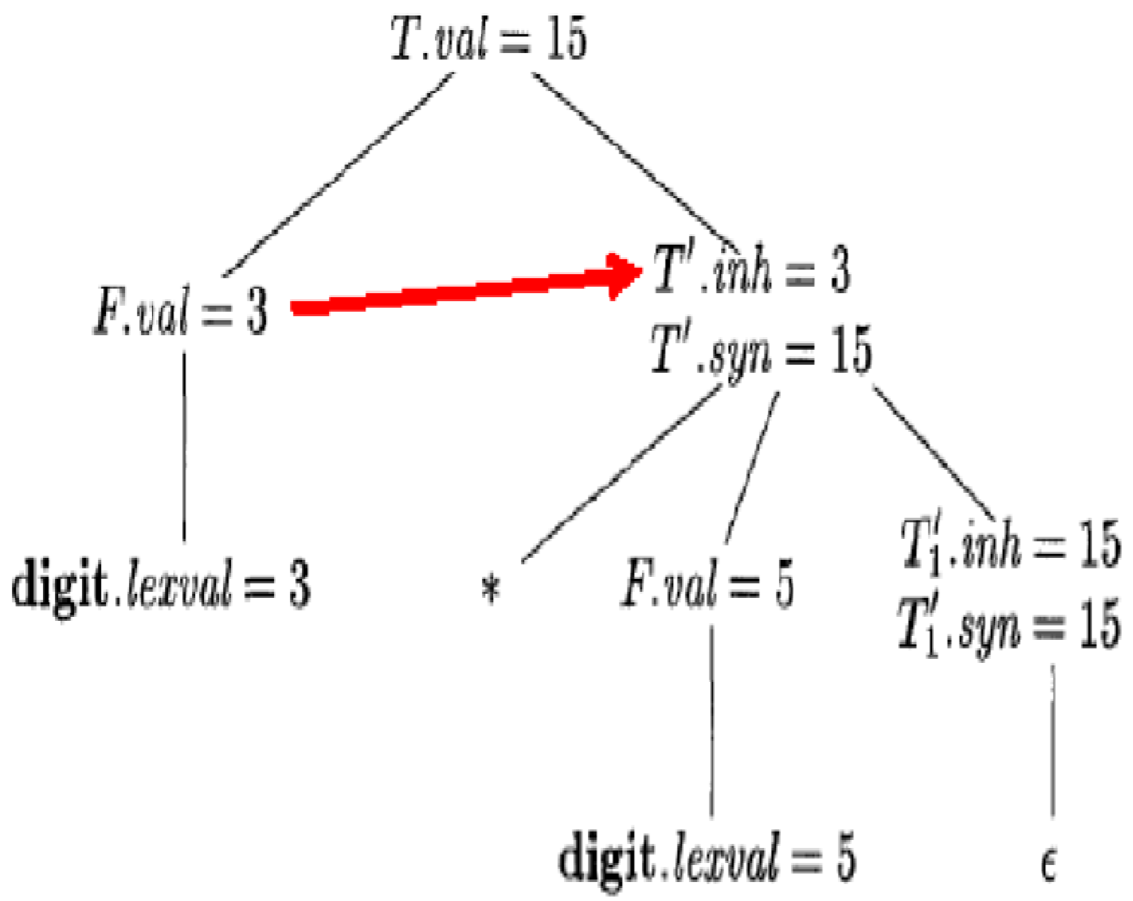


Annotated parse tree for $3 * 5$

Example 5.3(Cont…)

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

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

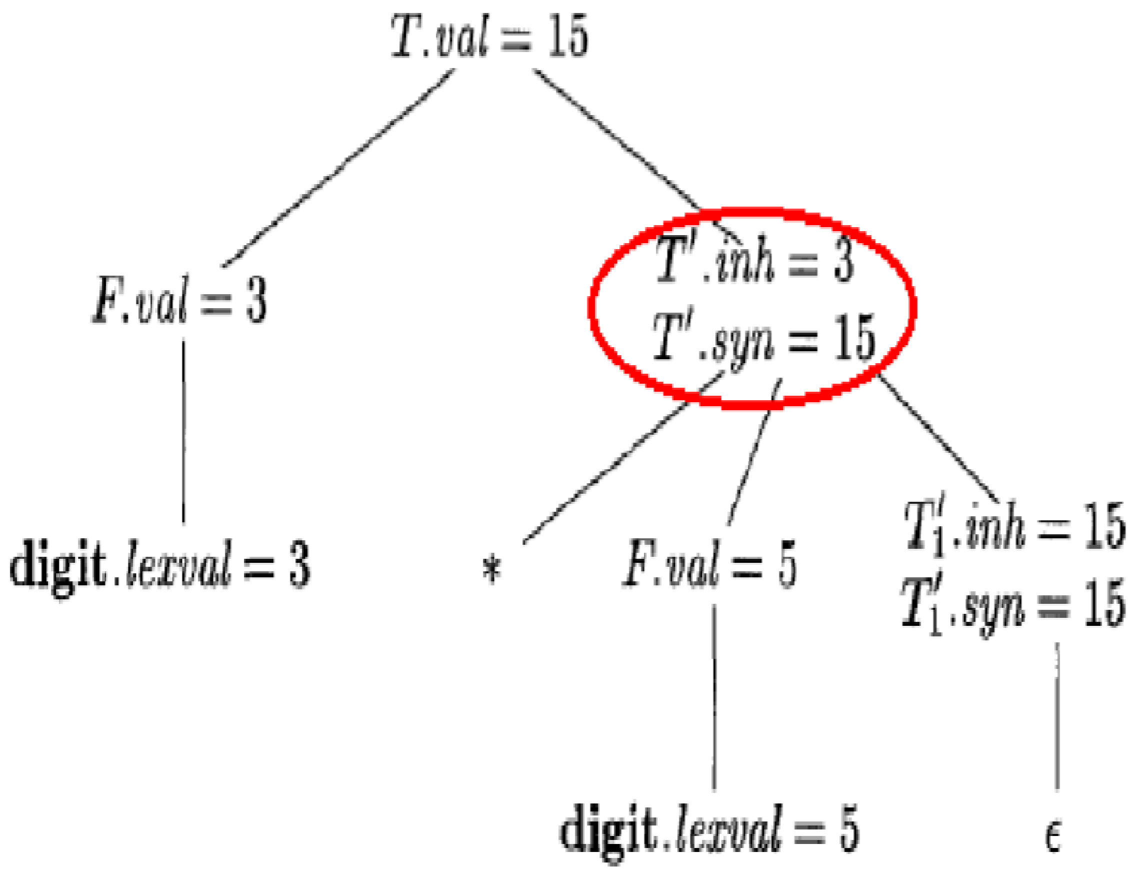


Annotated parse tree for $3 * 5$

Example 5.3(Cont...)

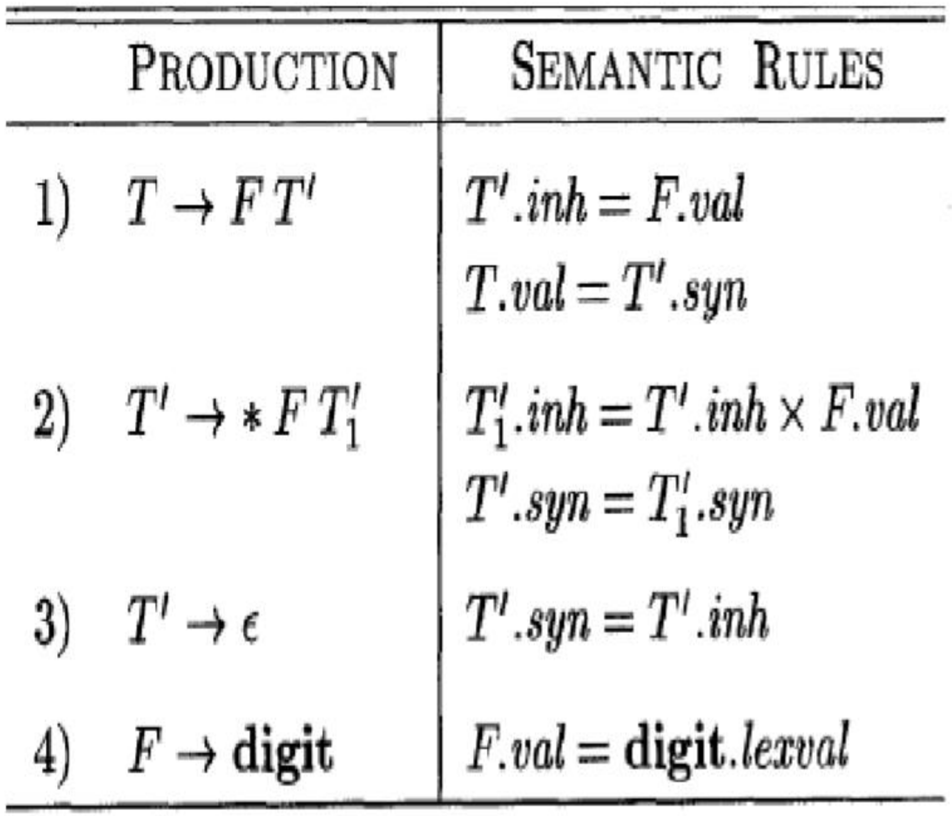
PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

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



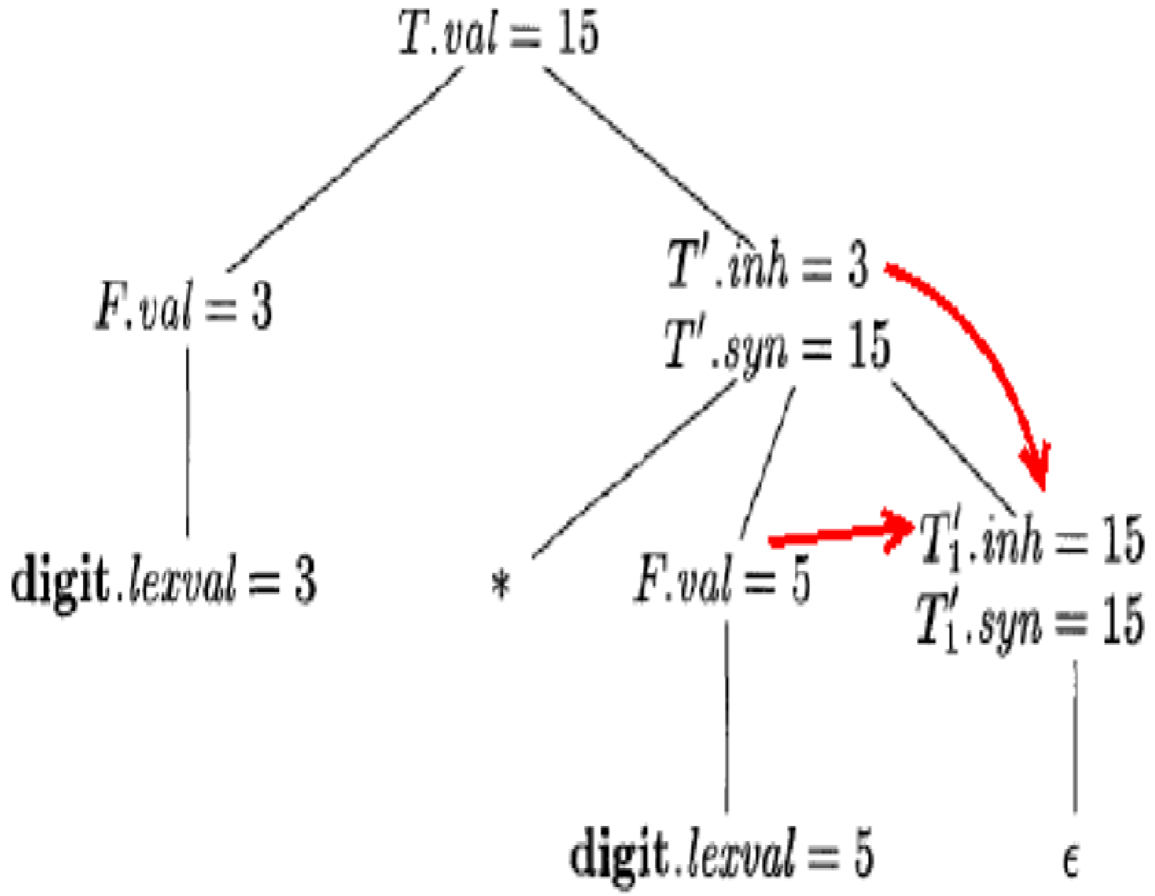
Annotated parse tree for $3 * 5$

Example 5.3(Cont…)



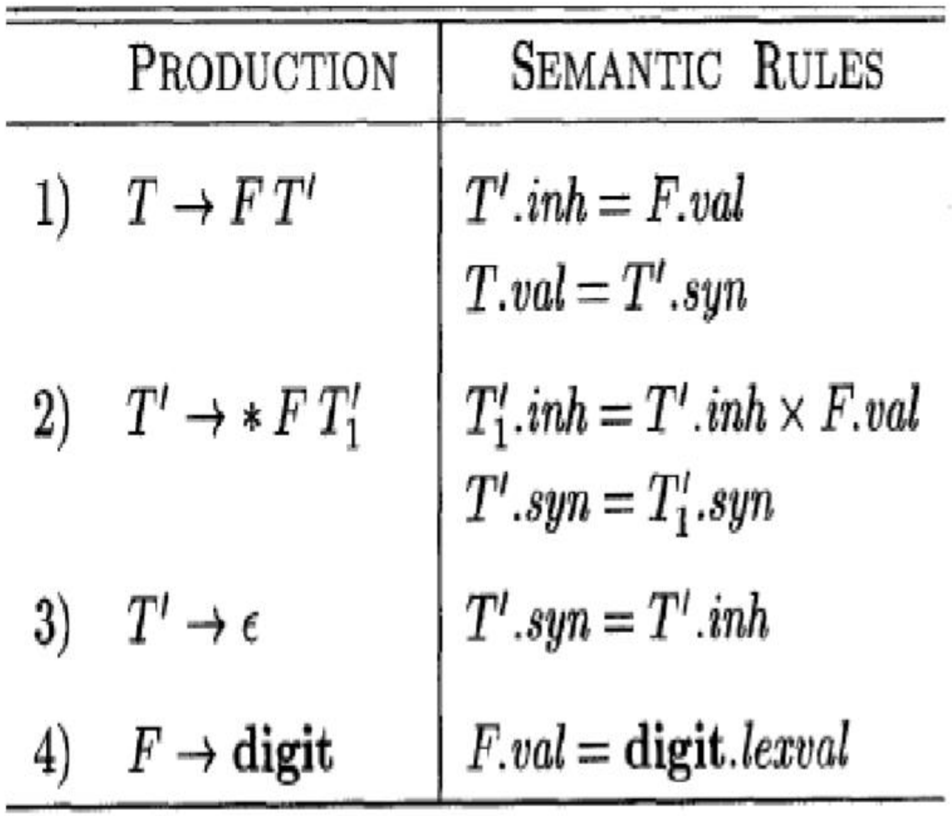
Production	Semantic Rules
1) $T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

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



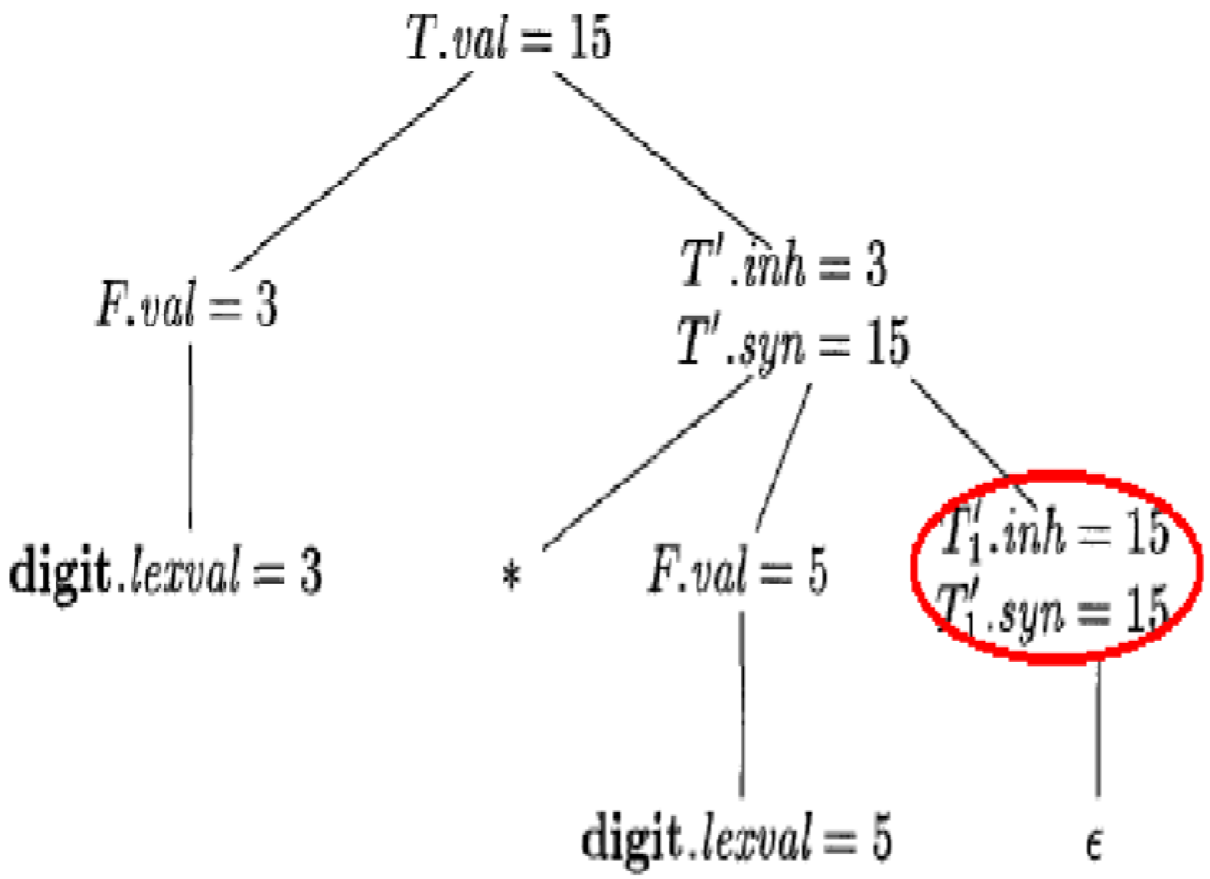
Annotated parse tree for $3 * 5$

Example 5.3(Cont…)



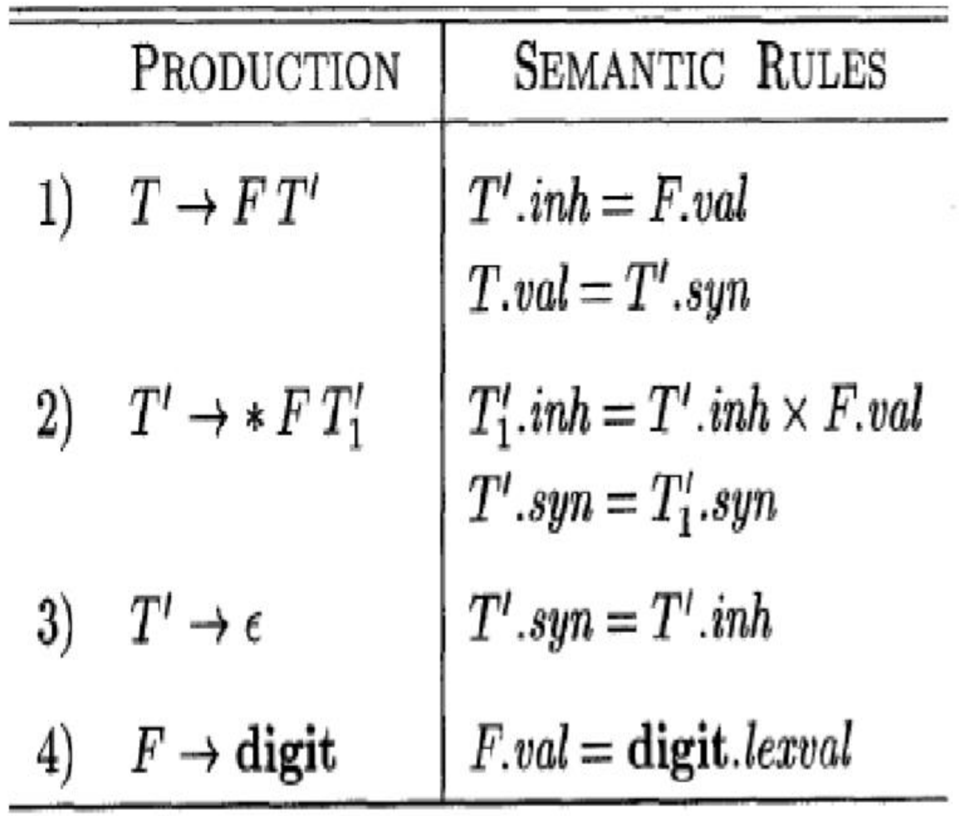
	Production	Semantic Rules
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * \, F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

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



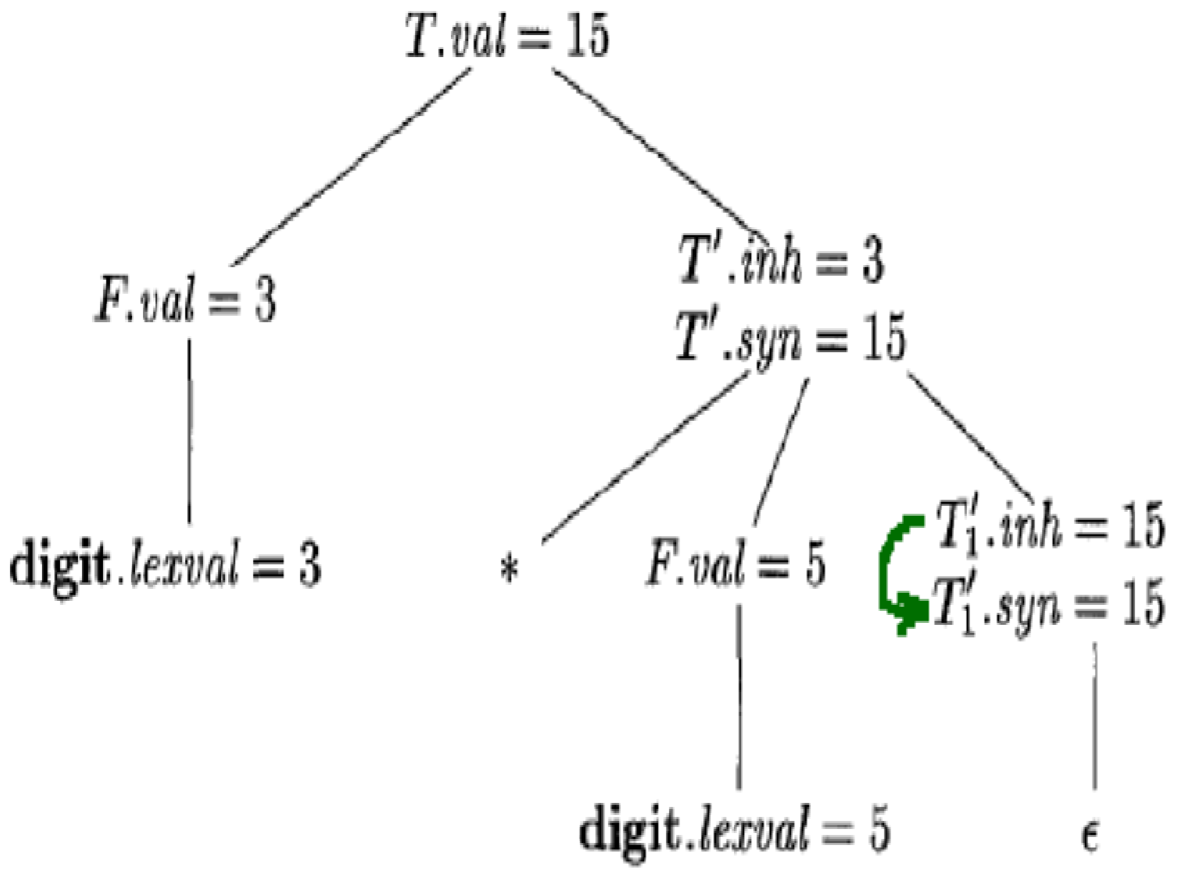
Annotated parse tree for $3 * 5$

Example 5.3(Cont…)



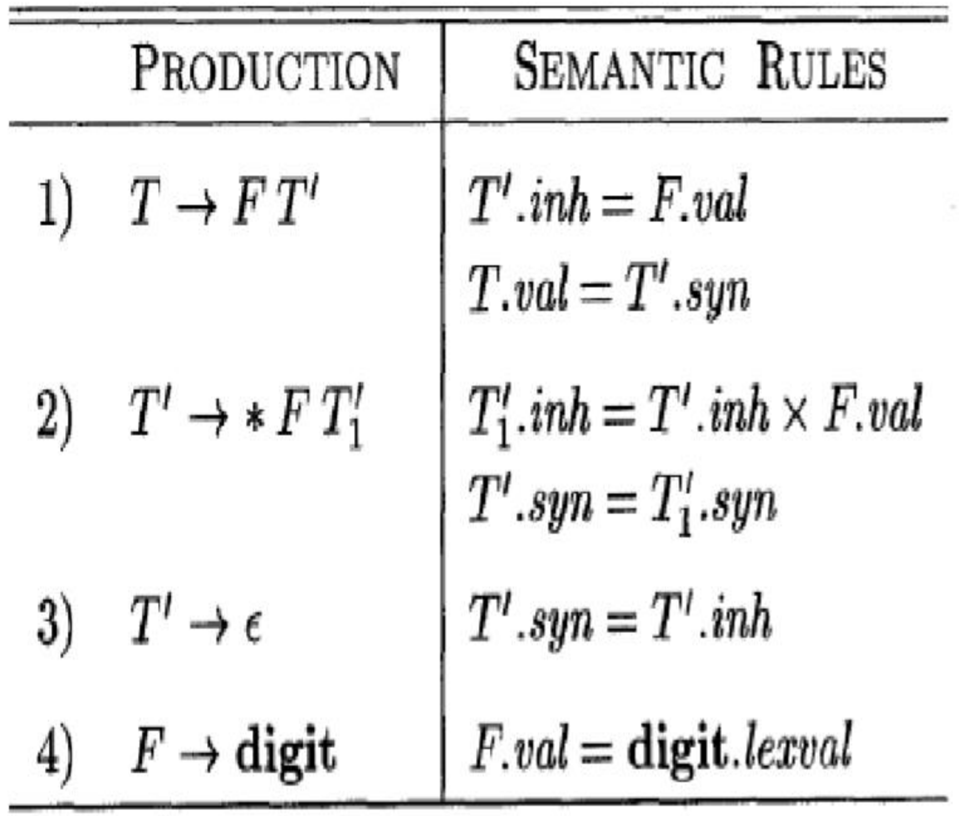
	PRODUCTION	SEMANTIC RULES
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

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



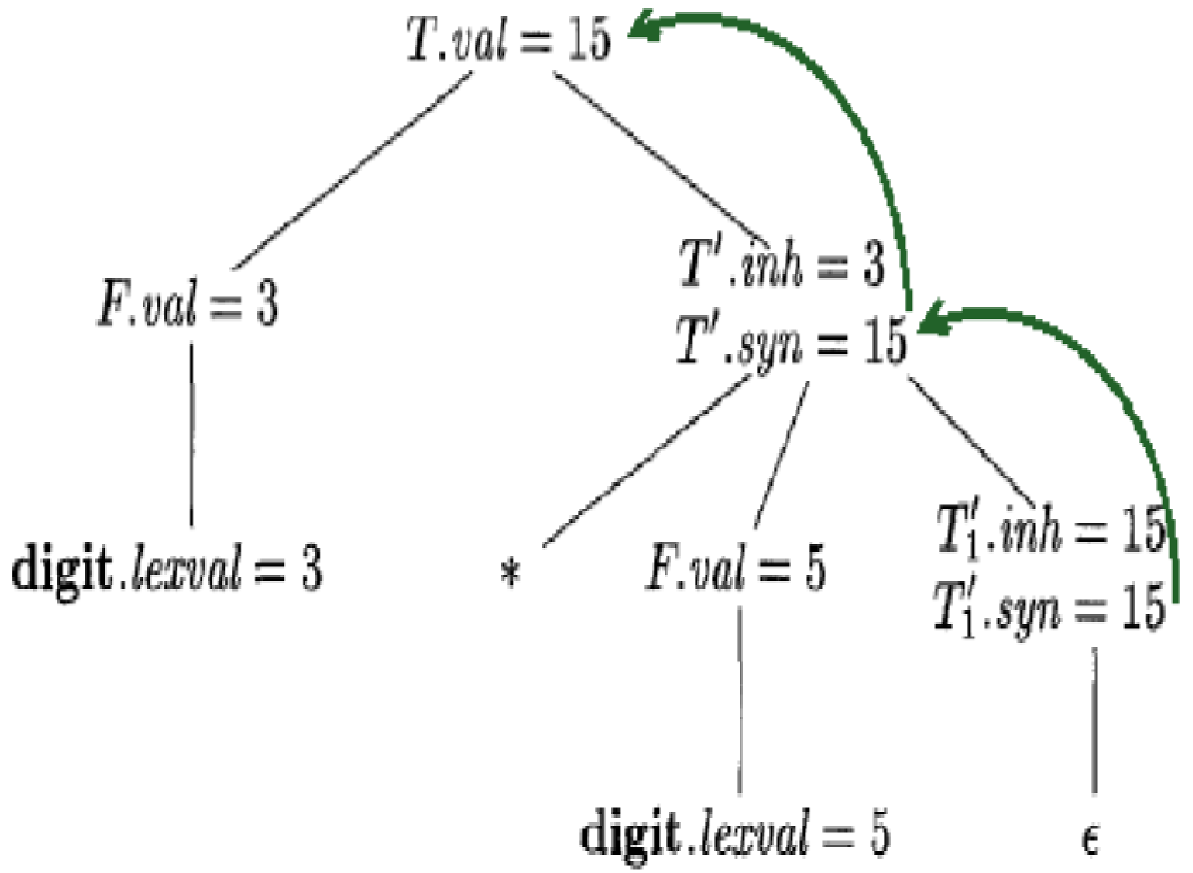
Annotated parse tree for $3 * 5$

Example 5.3(Cont…)



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

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

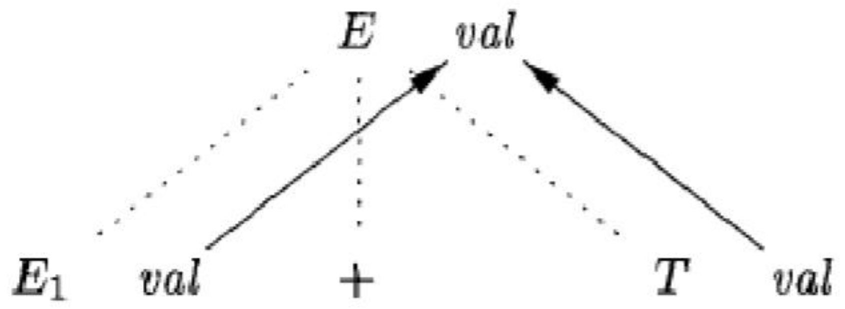


Annotated parse tree for $3 * 5$

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
- 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

Dependency Graphs(Cont…)



$E.val$ is synthesized from $E_1.val$ and $E_2.val$

- Suppose that a semantic rule associated with a production p defines the value of synthesized attribute E.val in terms of the value of E_1.val and T.val
- Then, the dependency graph has an edge from E_1.val to E.val

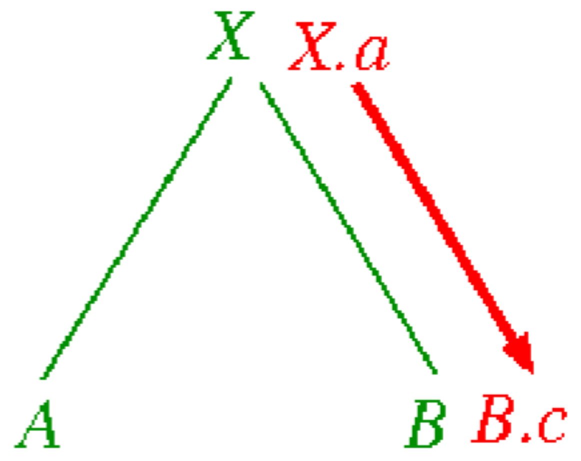
  and an edge from T.val to E.val

Dependency Graphs(Cont…)

- For each node N labeled E where production p is applied, create an edge to attribute val at N, from the attribute val at the left child of N corresponding to the instance of the symbol E_1 in the body of the production and another edge to attribute val at N, from the attribute val at the right child of N corresponding to the instance of the symbol T in the body of the production

Dependency Graphs(Cont…)

PRODUCTION	SEMANTIC RULE
$p : X \rightarrow AB$	$B.c = 2 \times X.a$



- 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

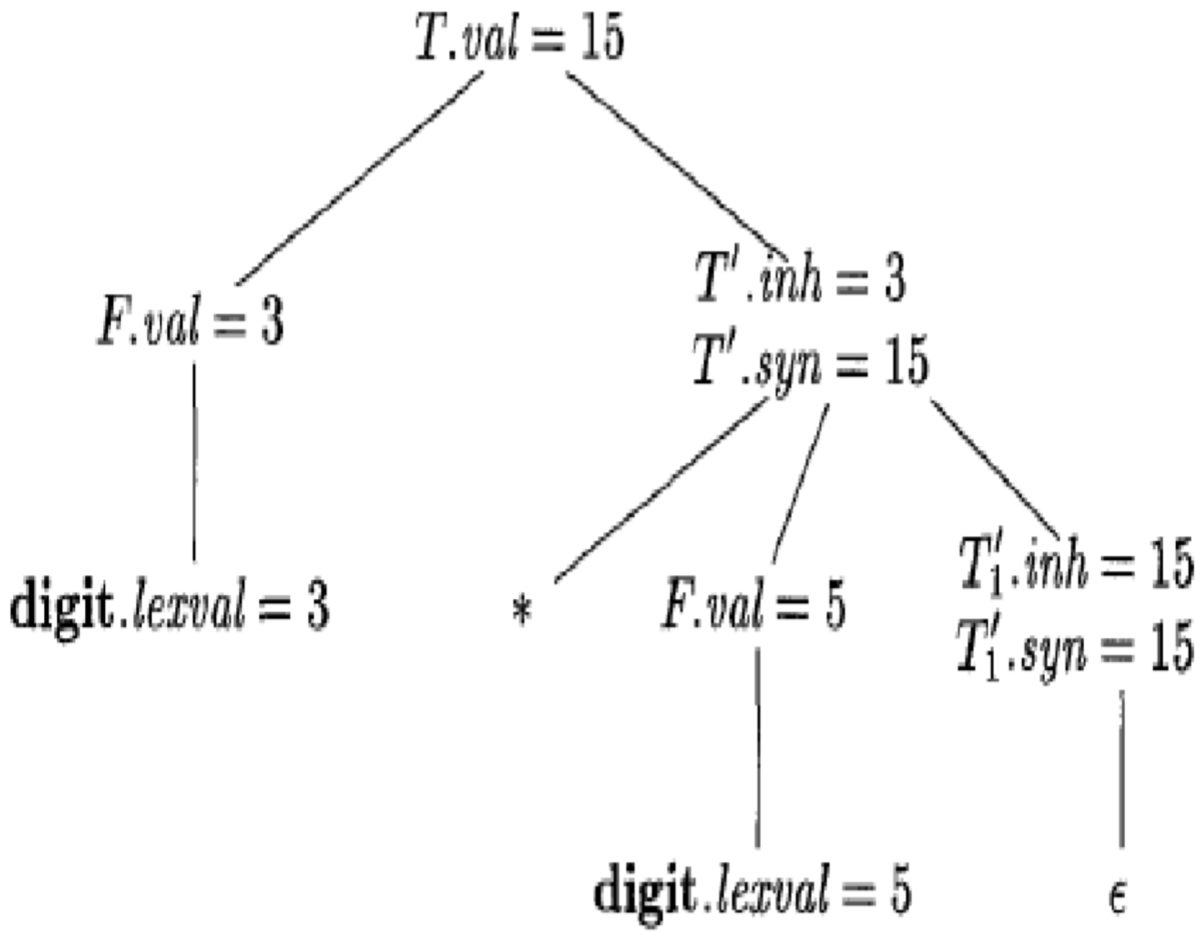
Dependency Graphs (Cont…)

- 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 of a Completely Dependency Graph

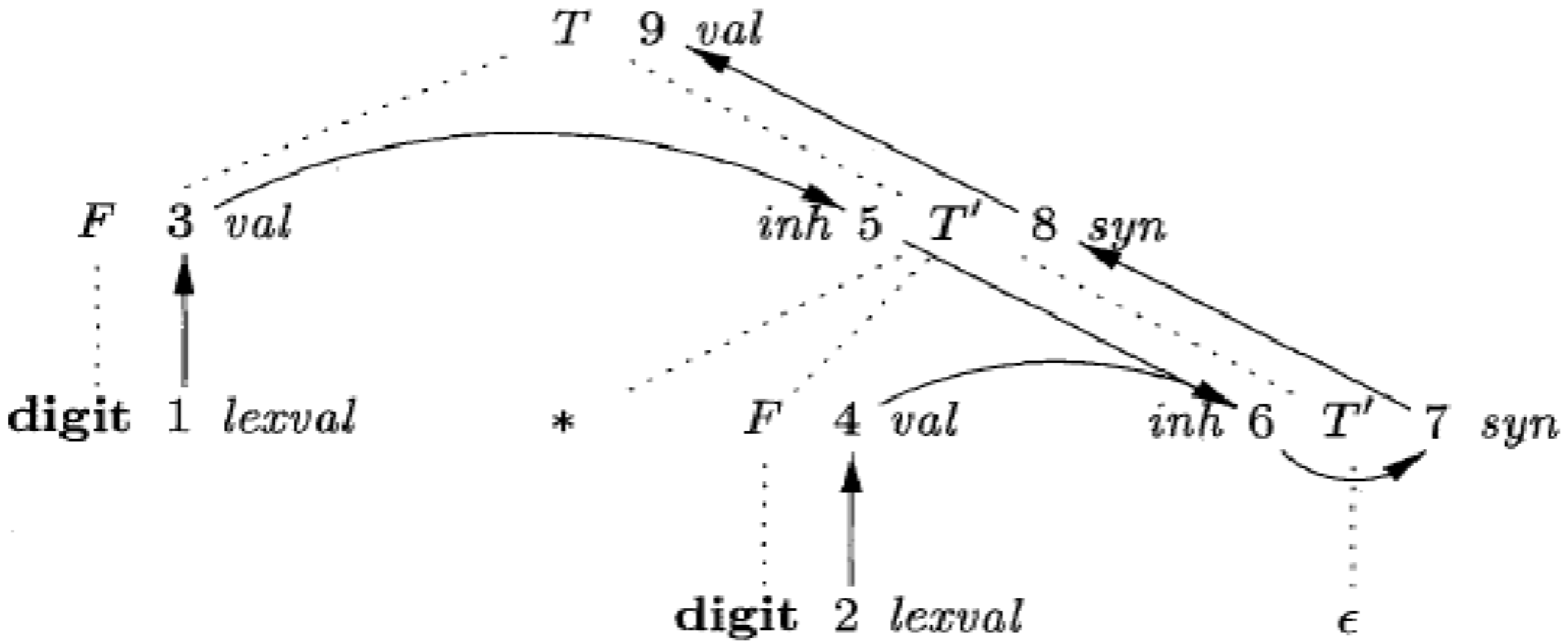
PRODUCTION	SEMANTIC RULES
1) $T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to * \, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

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



Annotated parse tree for $3 * 5$

Example of a Completely Dependency Graph



Dependency graph for the annotated parse tree

Ordering the Evaluation of Attributes (Topological Sort)

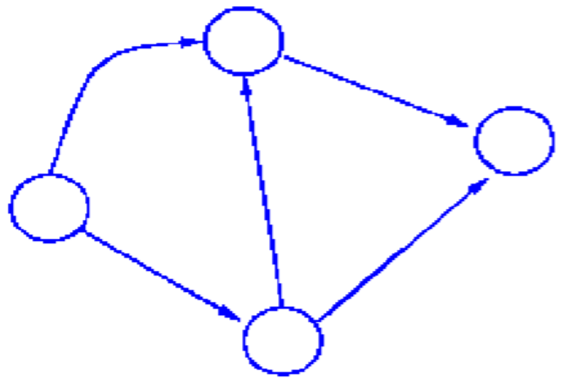
- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree
- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N
- Thus, the only allowable orders of evaluation are those sequences of nodes N_1, N_2, ..., 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

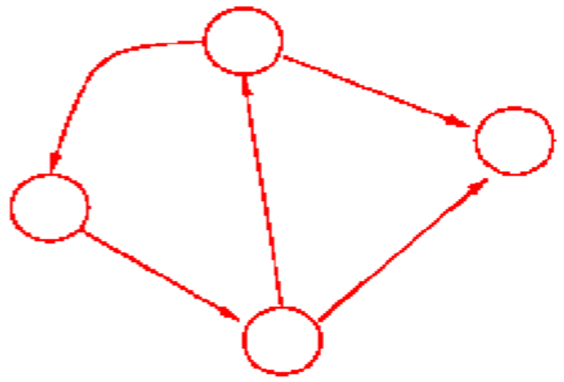
Ordering the Evaluation of Attributes(Cont…)

- 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

Ordering the Evaluation of Attributes(Cont…)

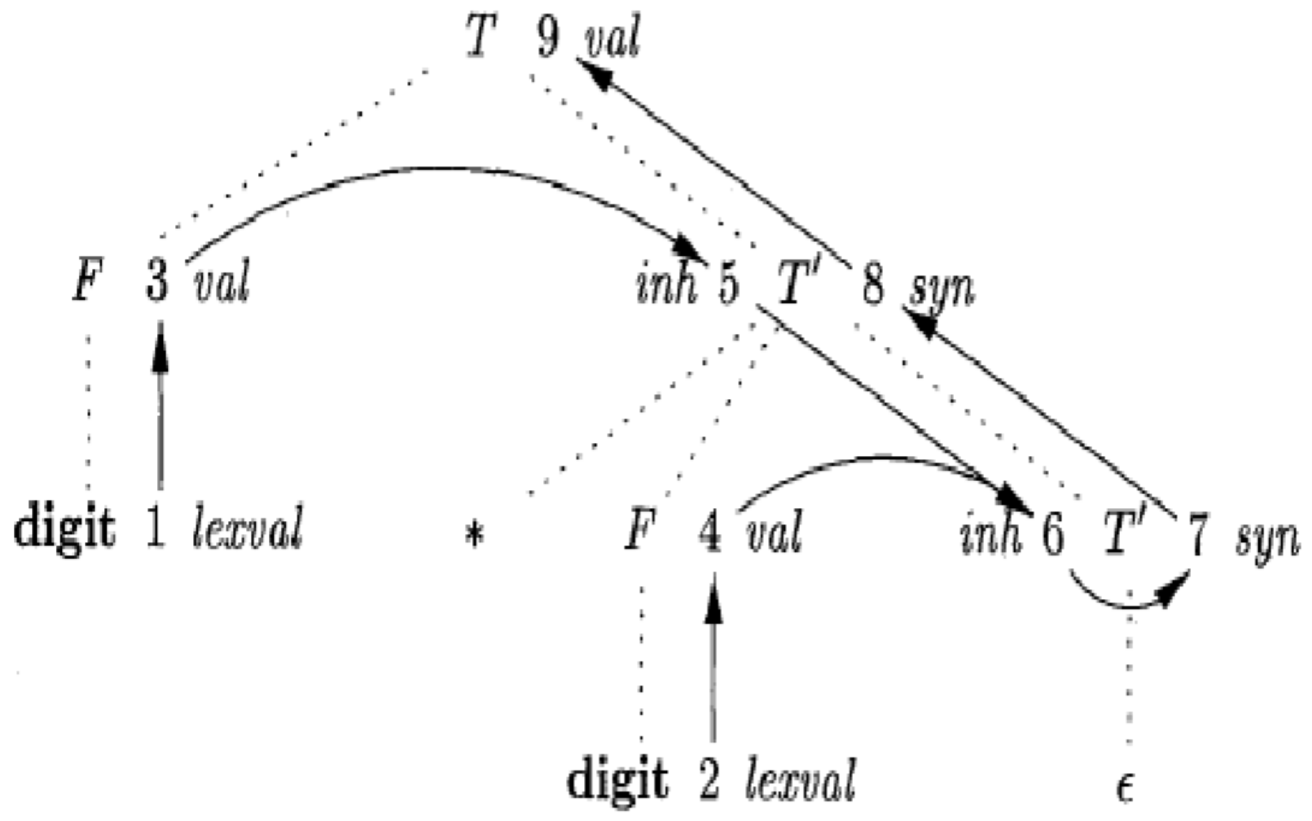
- To see why, since there are no cycles, we can surely find a node with no edge entering
- For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle





- Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes

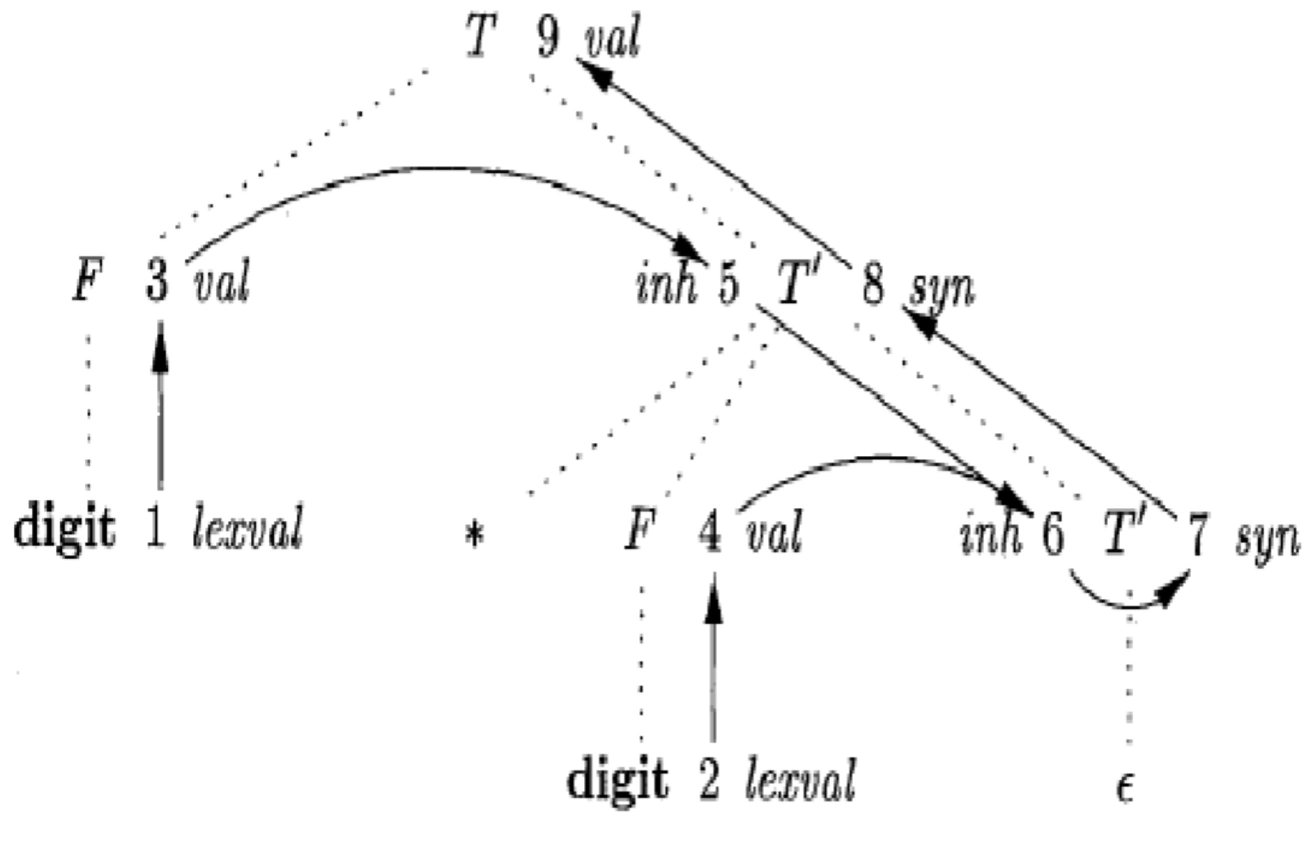
Example 5.5



Dependency graph for the annotated parse tree

- The dependency graph of figure has no cycles
- One topological sort is the order in which the nodes have already been numbered: 1,2,...,9

Example 5.5(Cont…)



Dependency graph for the annotated parse tree

- 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

Classes of SDD

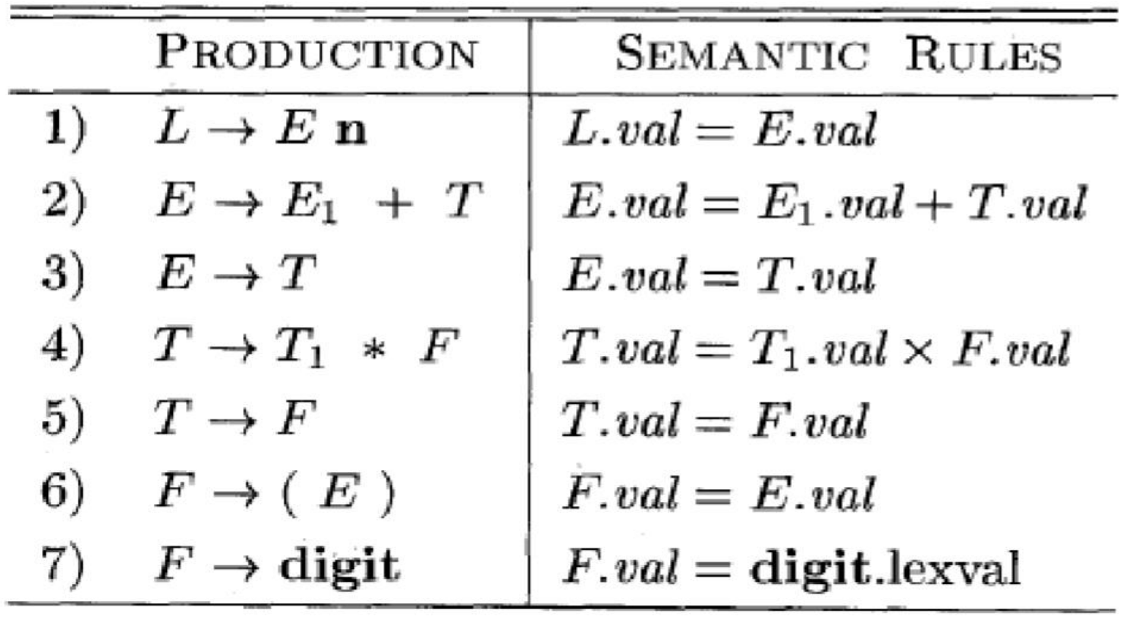
- From a given SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles
- In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles
- There are two classes of SDD
  - S-Attributed Definitions
  - L-Attributed Definitions
- Moreover, the two classes can be implemented efficiently in connection with top-down or bottom-up parsing

S-Attributed Definitions

- The first class is defined as follows:
  - An SDD is S-attributed if **every** attribute is synthesized

S-Attributed Definitions(Cont…)

- The SDD of figure is an example of an S-attributed definition
- Each attribute, L.val, E.val, T.val, and F.val is synthesized



	Production	Semantic Rules
1)	$L \to E$ **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)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

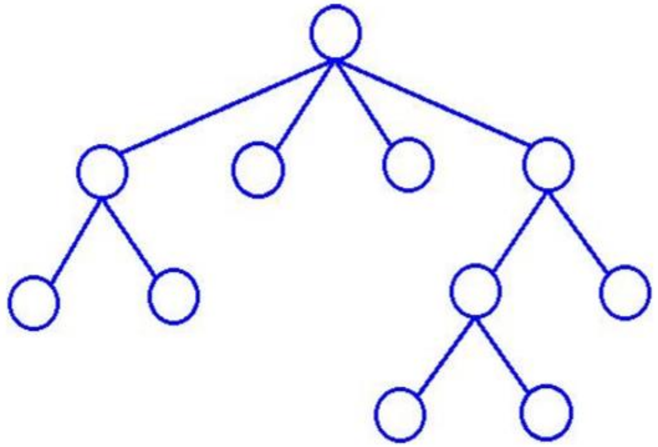
S-Attributed Definitions(Cont…)

- 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.

S-Attributed Definitions(Cont…)

- That is, we apply the function postorder, defined below, to the root of the parse tree,

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



S-Attributed Definitions(Cont…)

- 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

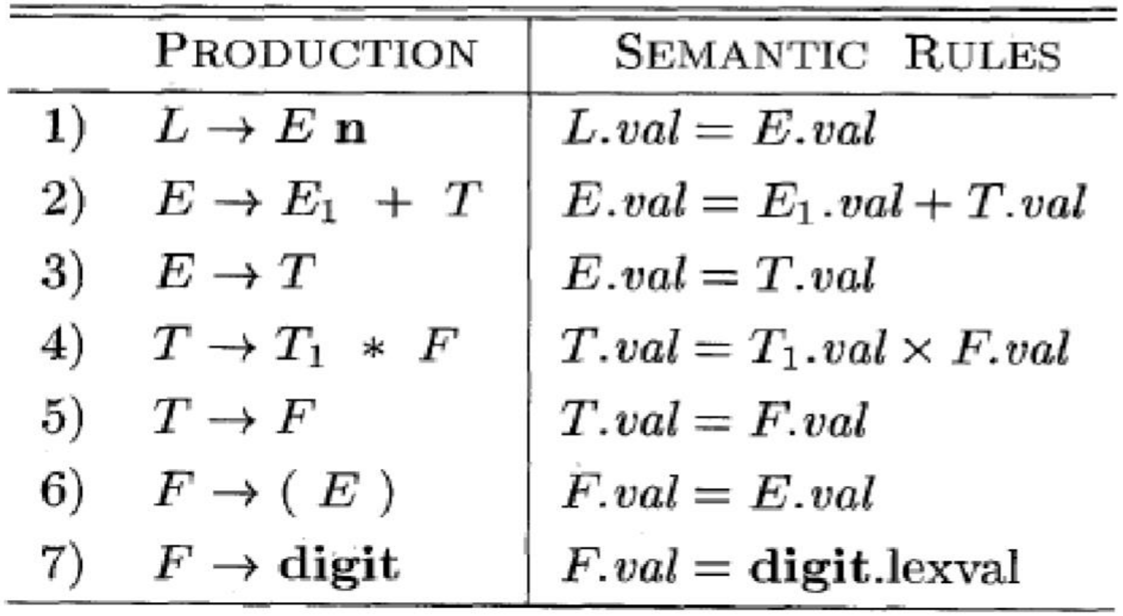
L-Attributed Definitions

- The second class of SDD’s is called 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”)
- More precisely, each attribute must be either
  - Synthesized, **or**
  - Inherited, but with limited rules

L-Attributed Definitions(Cont…)

- Inherited, but with the rules limited as follows
  - Suppose that there is a production $A \rightarrow X_1X_2 \ldots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production
  - Then the rule may use only:
    - ➢ Inherited attributes associated with the head A
    - ➢ 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
    - ➢ 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

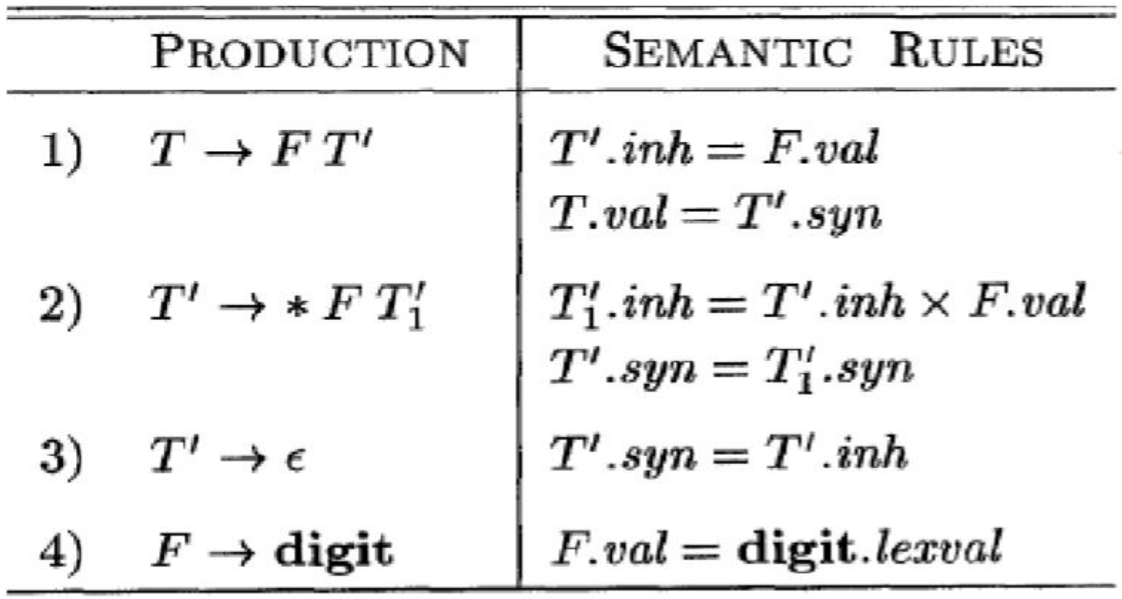


	Production	Semantic Rules
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)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

Syntax-directed definition of a simple desk calculator

- Is this SDD L-Attributed?
    - Yes. Because all of the grammar symbols have only synthesized attribute.

Example 5.8



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

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

- Is this SDD L-Attributed?

Example 5.8(Cont…)

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

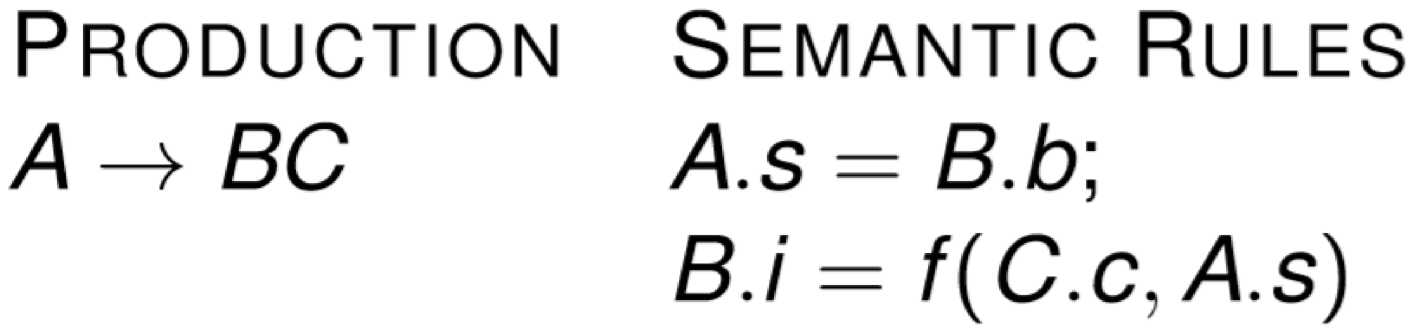
- The first of these rules defines the inherited attribute T´.inh using only F.val, and F appears to the left of T´ in the production body, as required
- The second rule defines T'_1.inh using the inherited attribute T´.inh associated with the head, and F.val, where F appears to the left of T'_1 in the production body

Example 5.8(Cont…)

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

- 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



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

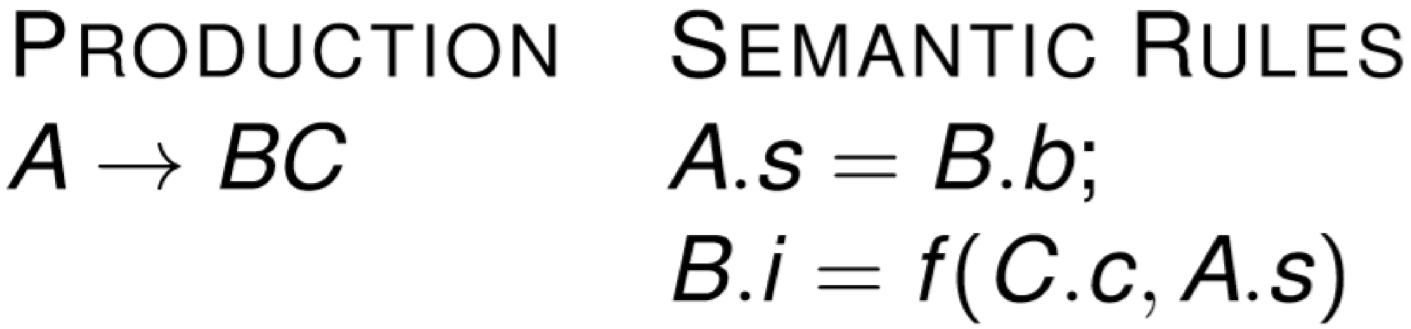
- Is this SDD S-Attributed or L-Attributed or Not?

Example 5.9(Cont…)

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

- The first rule, A.s = B.b, is a legitimate rule in either an S-attributed or L-attributed SDD
- It defines a synthesized attribute A.s in terms of an attribute at a child (that is, a symbol within the production body)

Example 5.9(Cont…)



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

- The second rule defines an inherited attribute B.i, so the entire SDD cannot be S-attributed
- Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute C.c is used to help define B.i, and C is to the right of B in the production body

Example 5.9(Cont…)

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

- While attributes at siblings in a parse tree may be used in L-attributed SDD’s, they must be to the left of the symbol whose attribute is being definedFurther, although the rule is legal, the SDD cannot be L-attributed, because the attribute C.c is used to help define B.i, and C is to the right of B in the production body

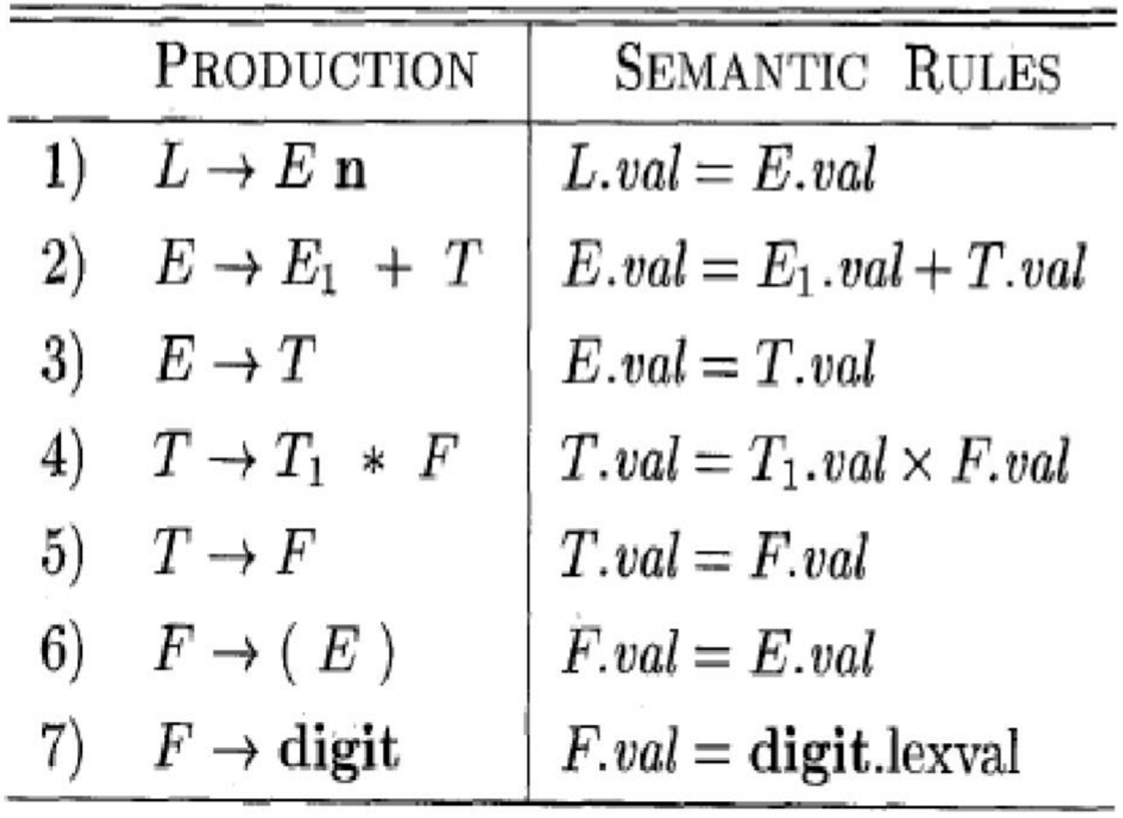
Semantic Rules with Controlled Side Effects

- In practice, translations involve side effects:
  - a desk calculator might print a result
  - a code generator might enter the type of an identifier into a symbol table
- With SDD’s, we strike a balance between attribute grammars and translation schemes
- 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

Semantic Rules with Controlled Side Effects(Cont..)

- We shall control side effects in SDD’s in one of the following ways:
  - 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

Semantic Rules with Controlled Side Effects(Cont..)



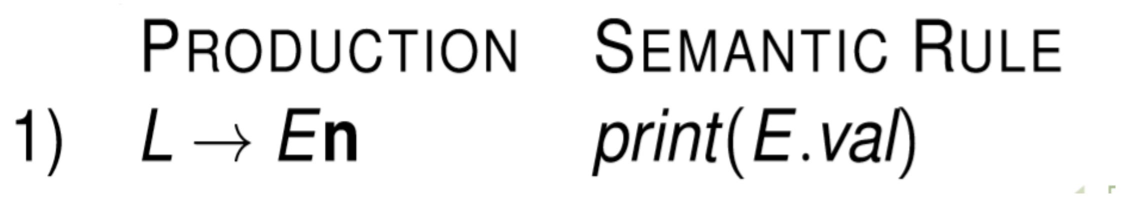
	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax-directed definition of a simple desk calculator

- As an example of an incidental side effect, let us modify the desk calculator 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)	$L \to E\mathbf{n}$	$print(E.val)$

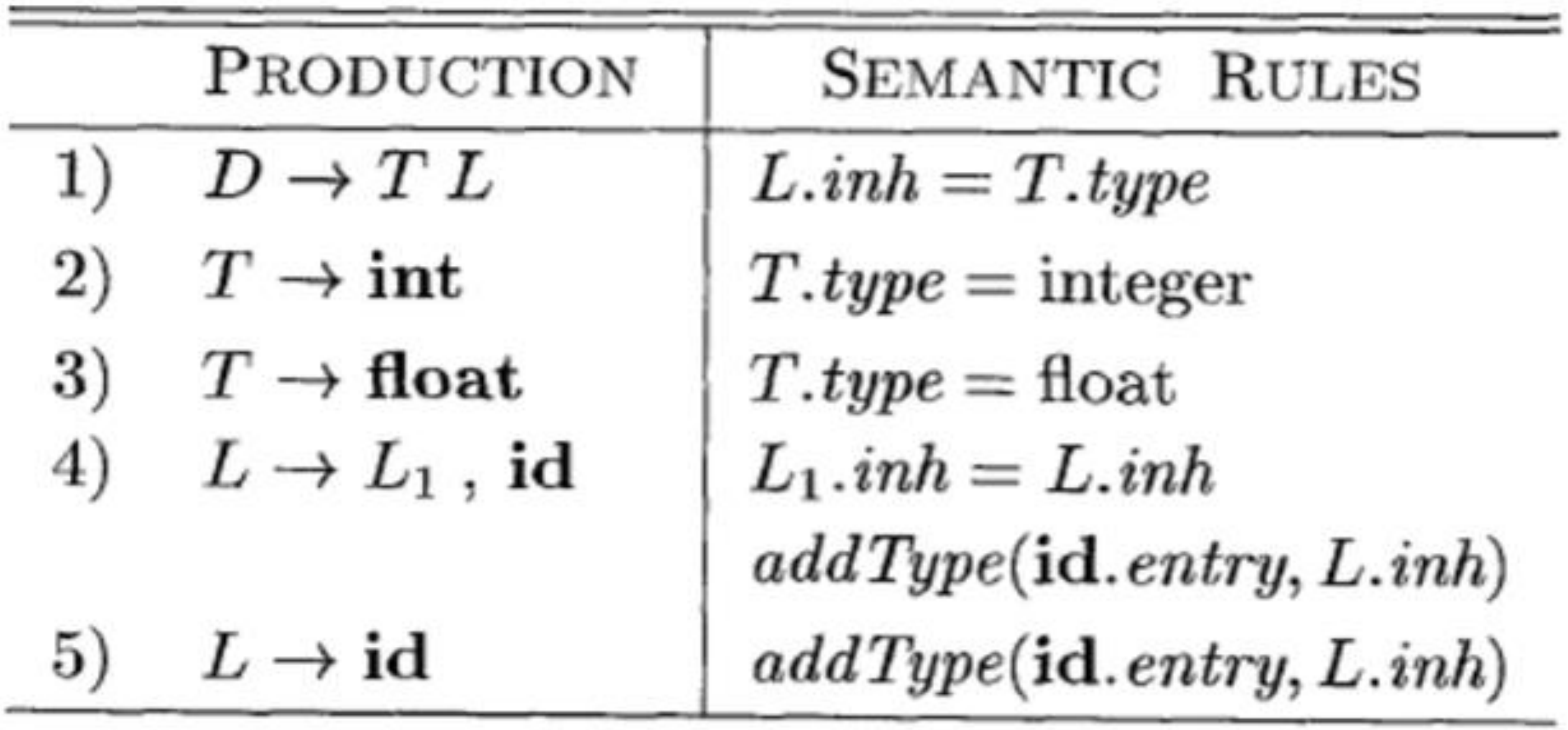
Semantic Rules with Controlled Side Effects(Cont..)



	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



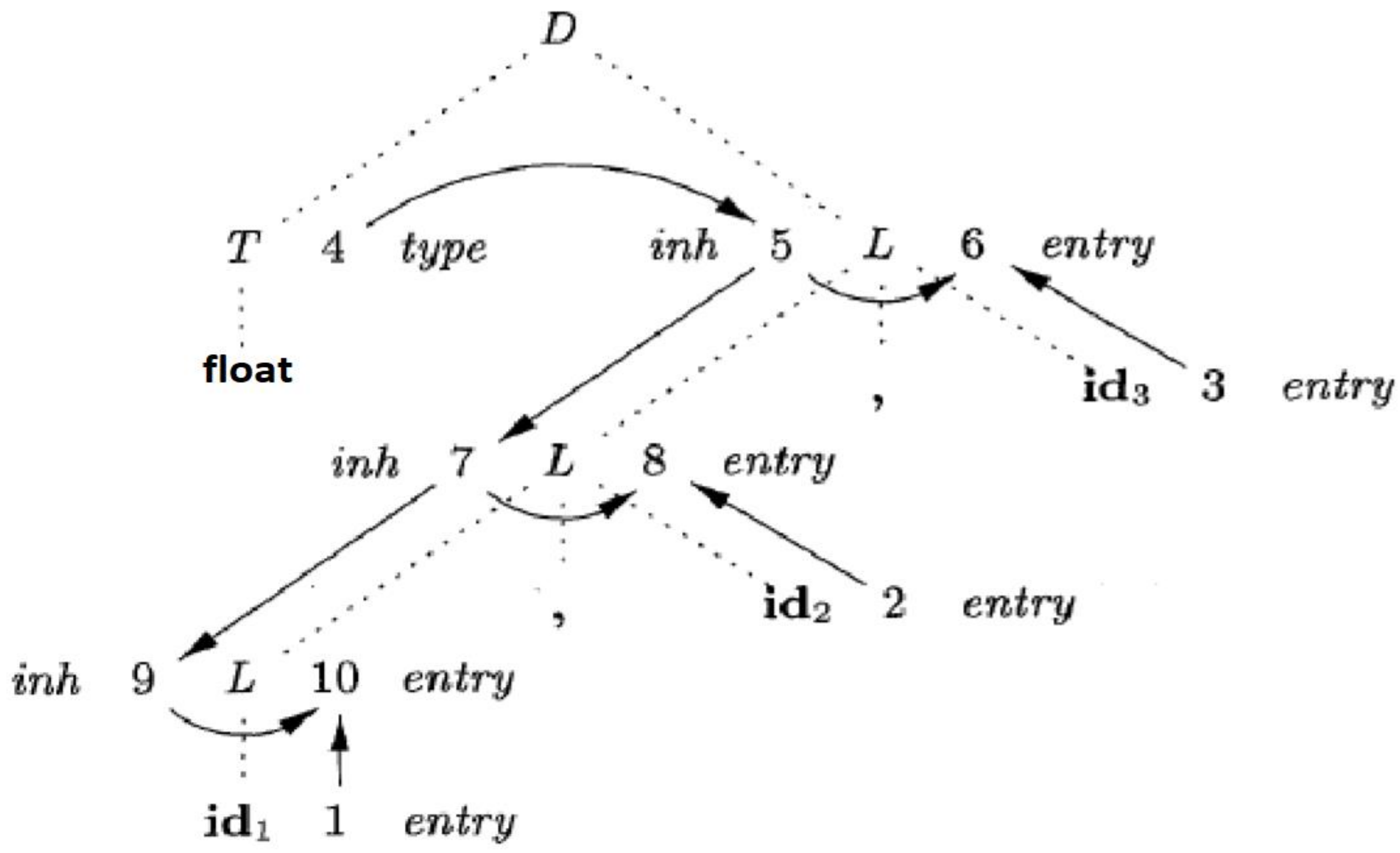
	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$
		$addType($**id**$.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType($**id**$.entry, L.inh)$

Syntax-directed definition for simple type declarations

Example 5.10(Cont…)

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

Syntax-directed definition for simple type declarations



float id$_1$, id$_2$, id$_3$

Applications of Syntax-Directed Translation

- The syntax-directed translation techniques will be applied to type checking and intermediate-code generation
- Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree
- 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

Construction of Syntax Trees

- 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 E1 +E2 has label + and two children representing the subexpressions E1 and E2

Construction of Syntax Trees(Cont…)

- 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

Construction of Syntax Trees(Cont…)

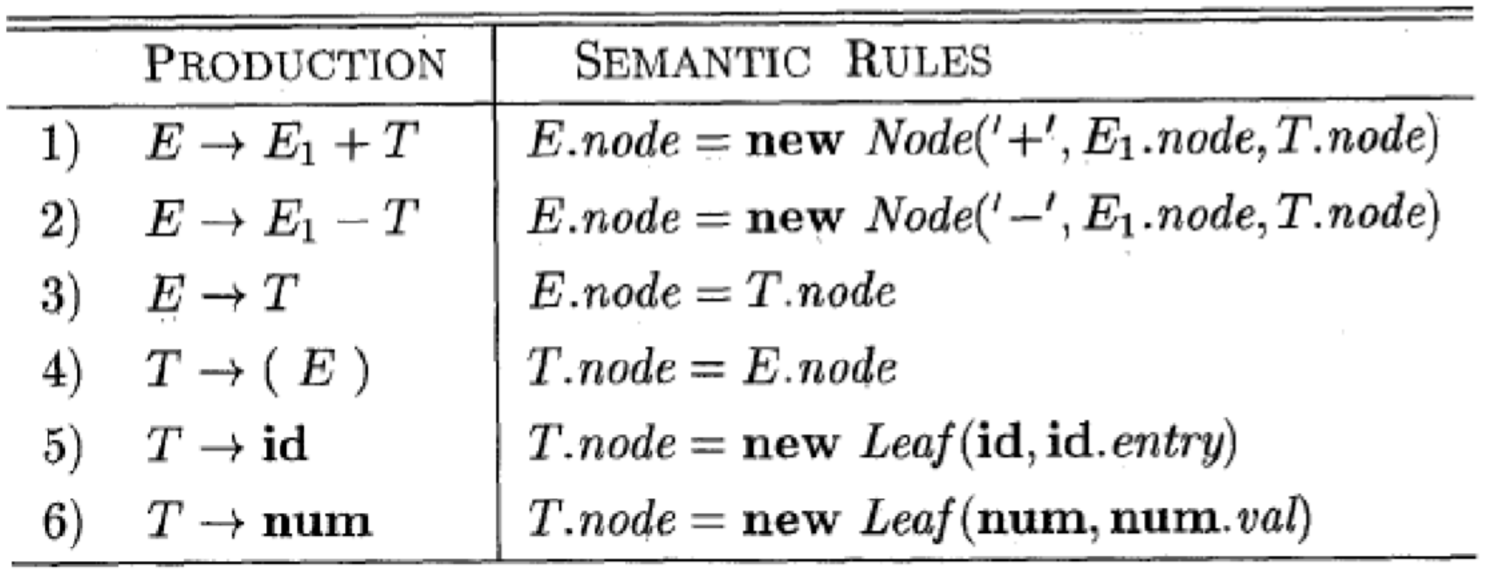
- If the node is a leaf, an additional field holds the lexical value for the leaf
- A constructor function Leaf(op,val) creates a leaf object
- Alternatively, if nodes are viewed as records, then Leaf returns a pointer to a new record for a leaf

Construction of Syntax Trees(Cont…)

- 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,c1,c2,...,ck) creates an object with first field op and k additional fields for the k children c1,...,ck

Example 5.11

The S-attributed definition in figure constructs syntax trees for a simple expression grammar involving only the binary operators + and −



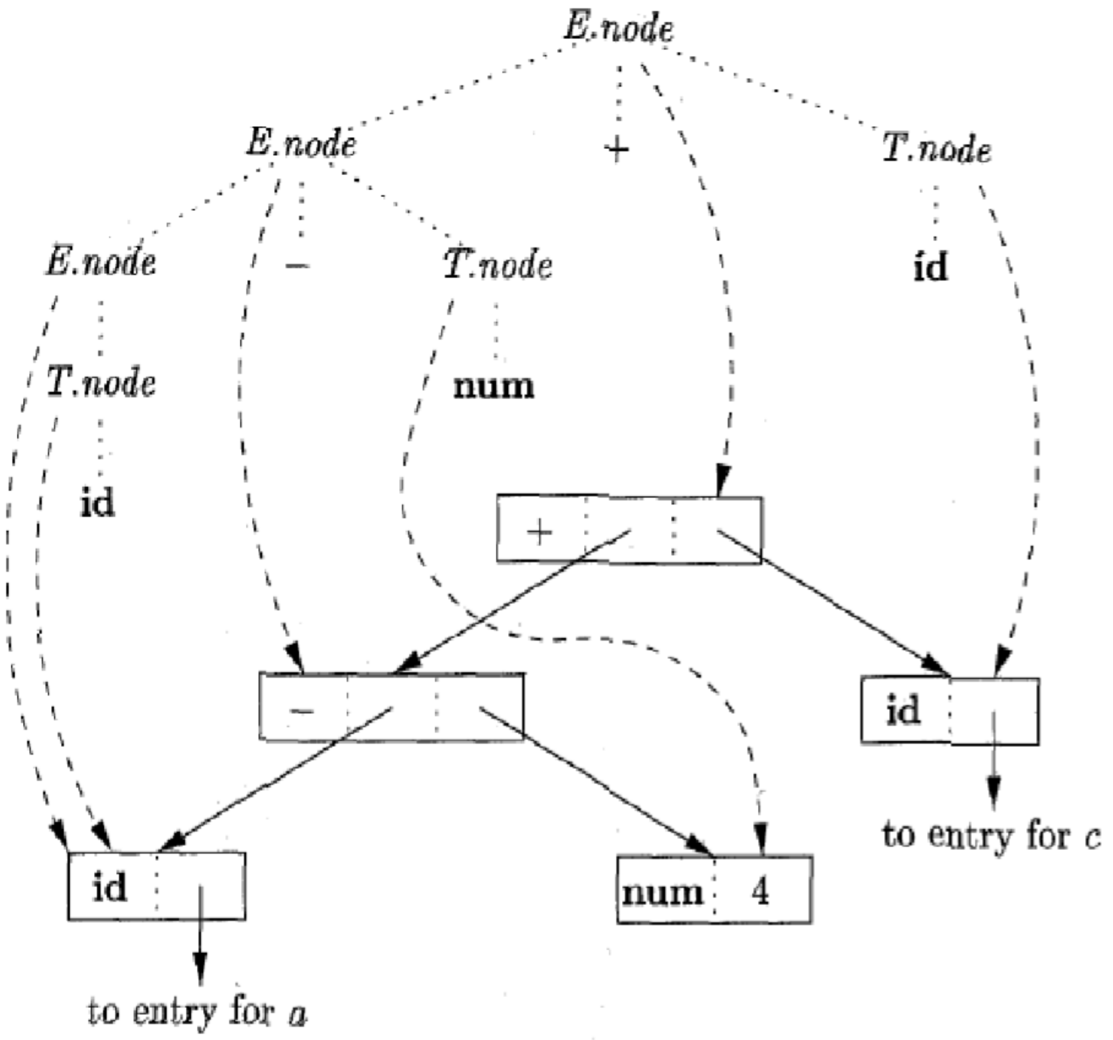
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$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

Constructing syntax trees for simple expressions

Example 5.11(Cont…)

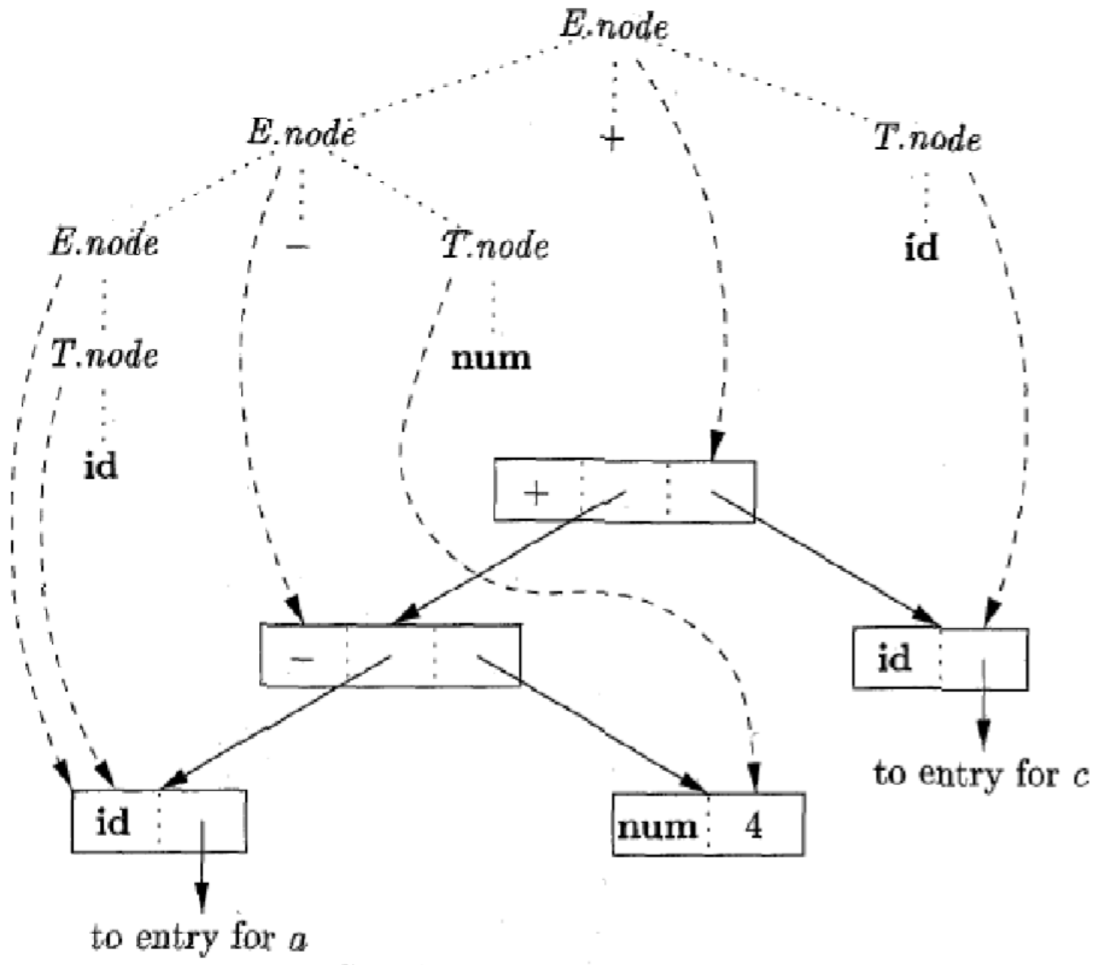
	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** $Leaf($**id**$,$ **id**$.entry)$
6)	$T \to$ **num**	$T.node =$ **new** $Leaf($**num**$,$ **num**$.val)$

Constructing syntax trees for simple expressions



Syntax tree for $a - 4 + c$

Example 5.11(Cont…)



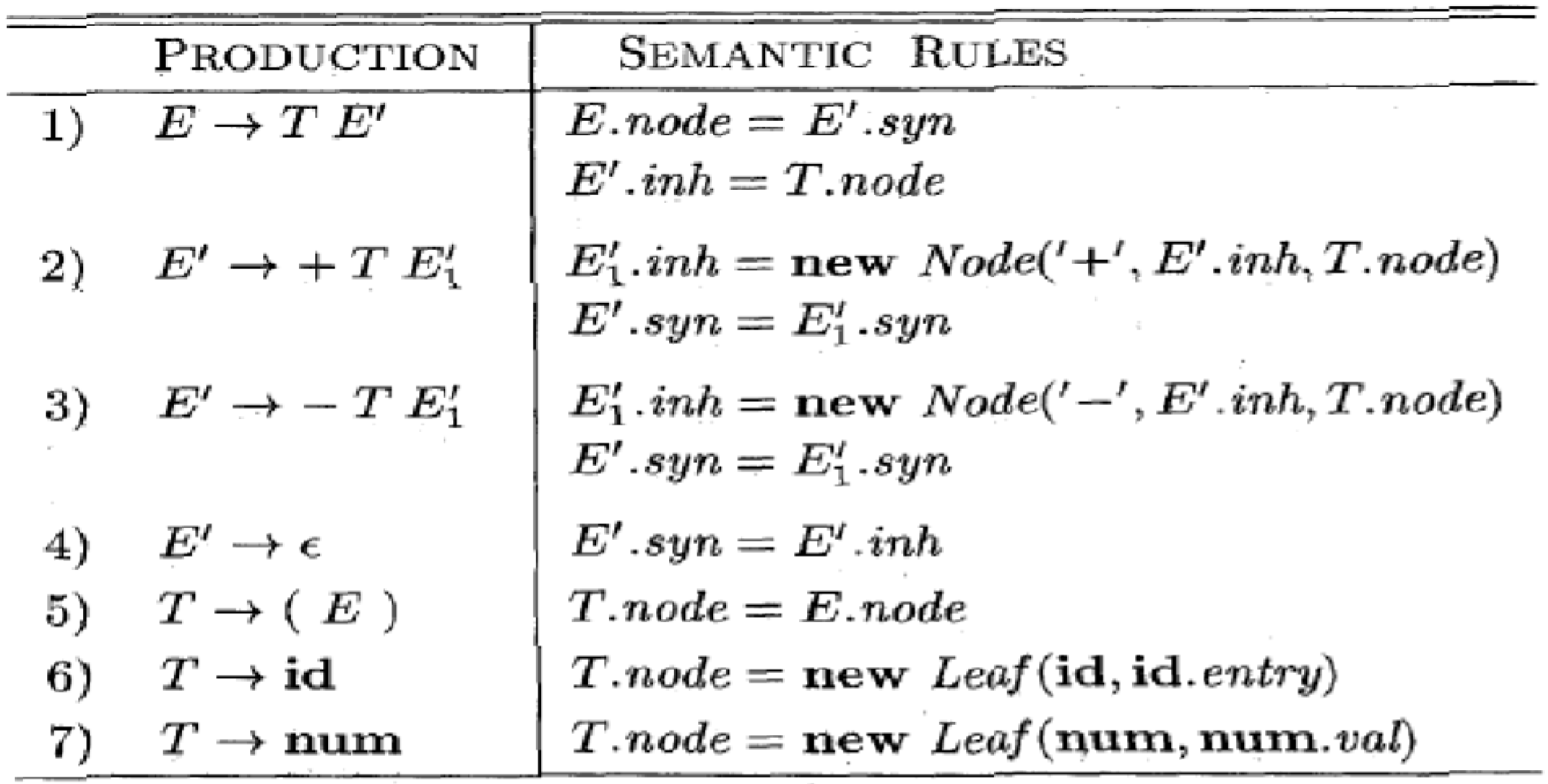
Syntax tree for $a - 4 + c$

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 $a - 4 + c$

Example 5.12

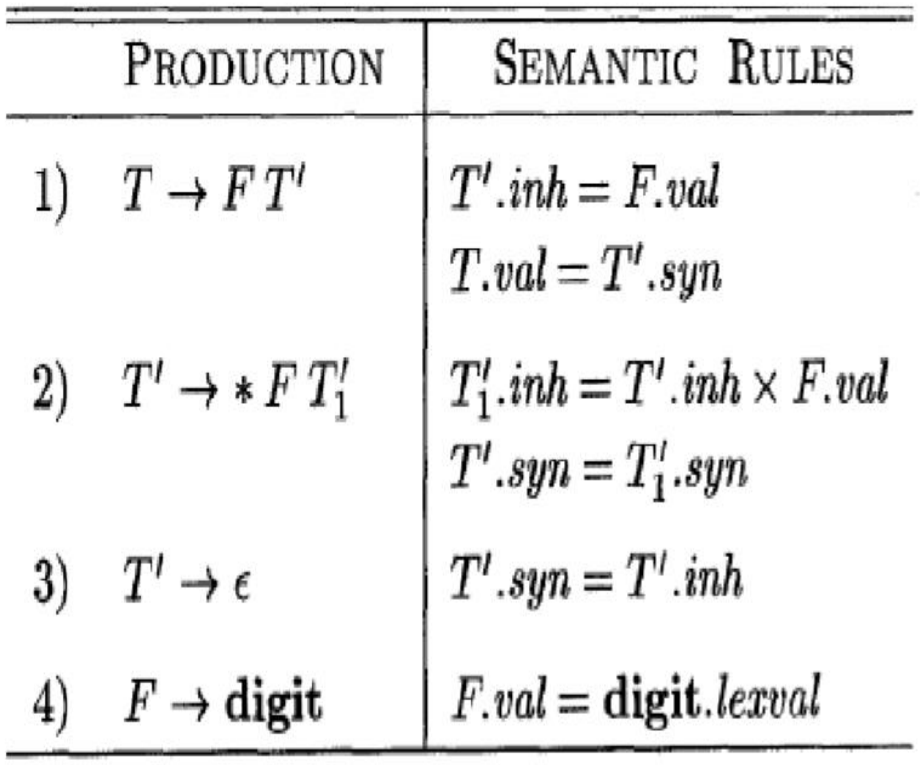
The L-attributed definition in this example performs the same translation as the S-attributed definition in the previous one



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

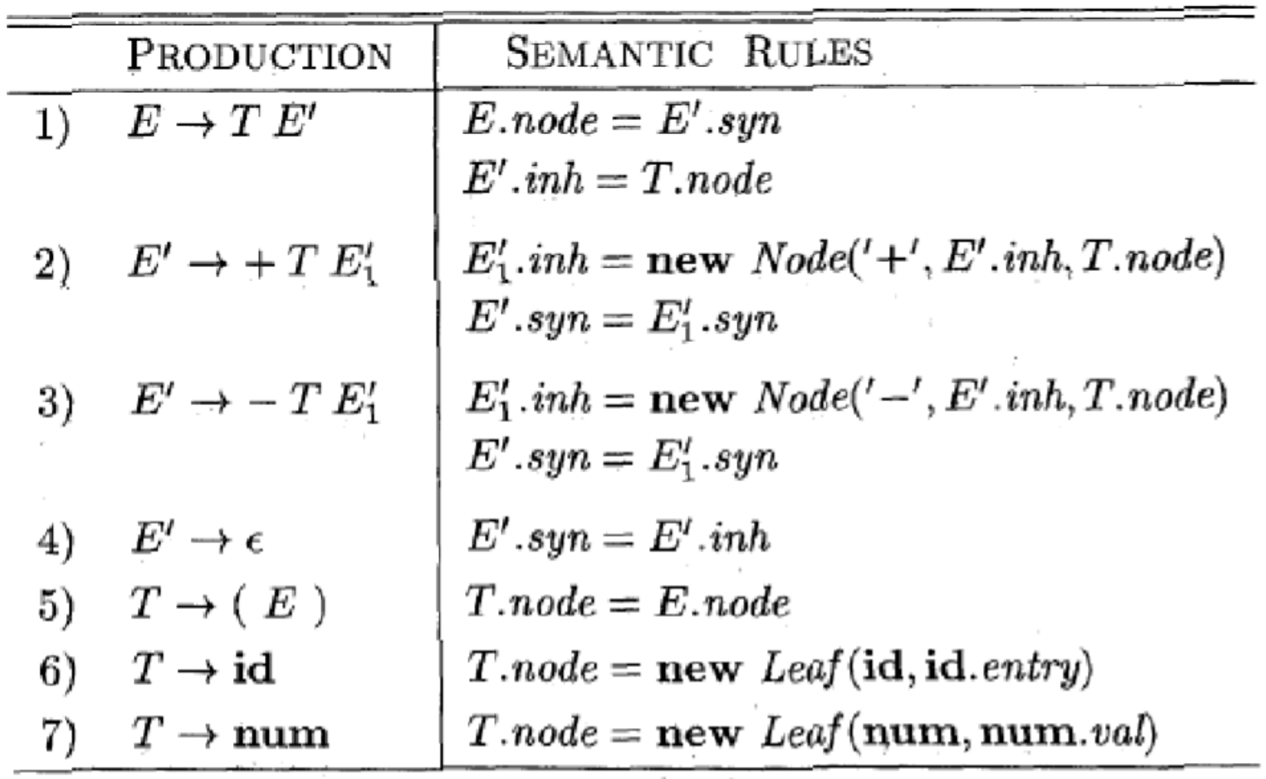
Constructing syntax trees during top-down parsing

Comparison of Example 5.3 and Example 5.12



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

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



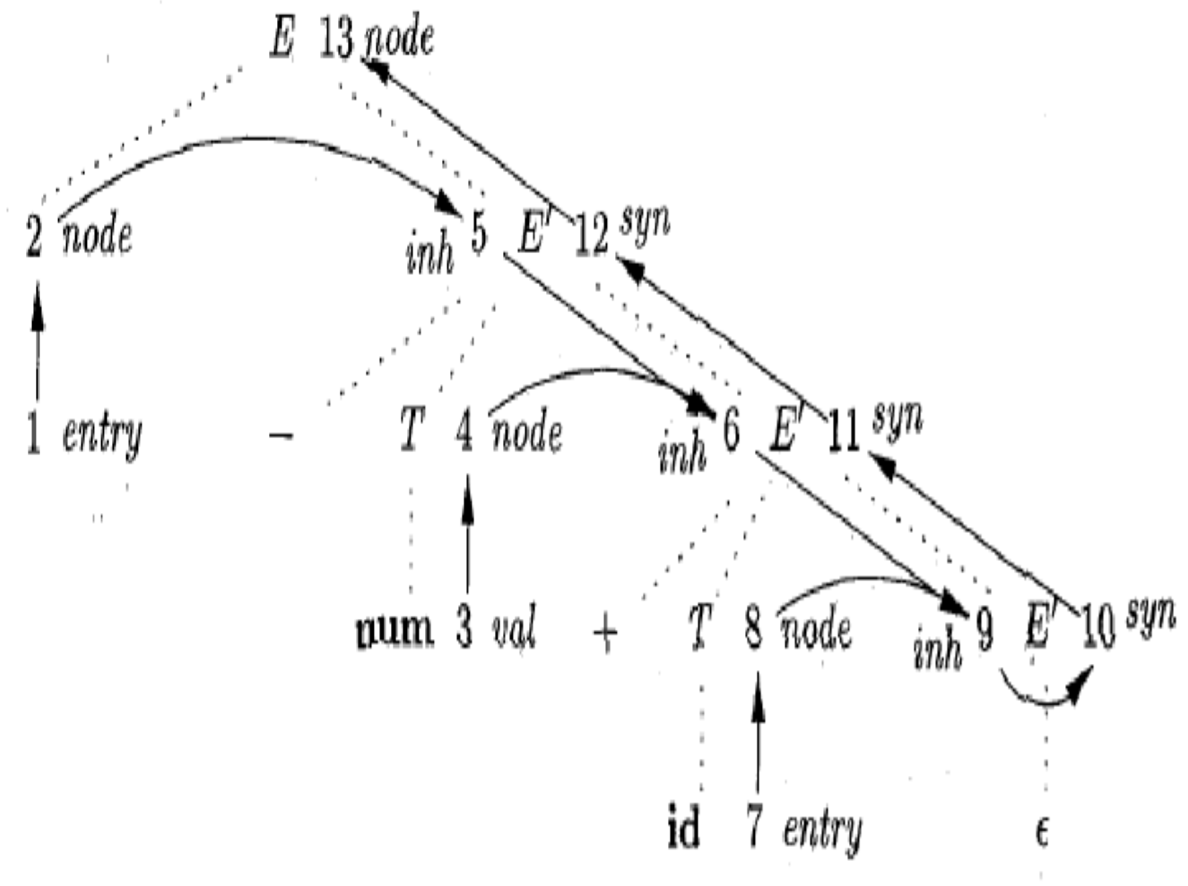
	PRODUCTION	SEMANTIC RULES
1)	$E \to T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \to +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \to -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \to \epsilon$	$E'.syn =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

Constructing syntax trees during top-down parsing

Example 5.12(Cont…)

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

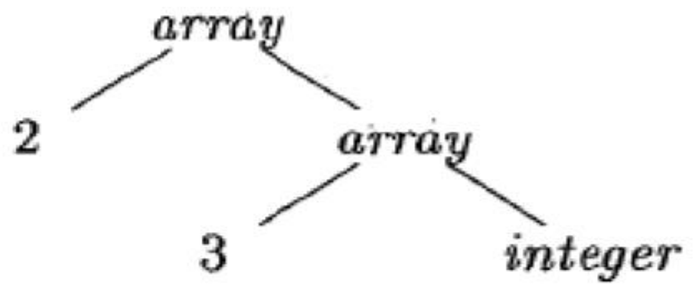
Constructing syntax trees during top-down parsing



Dependency graph for $a - 4 + c$,

Example 5.13

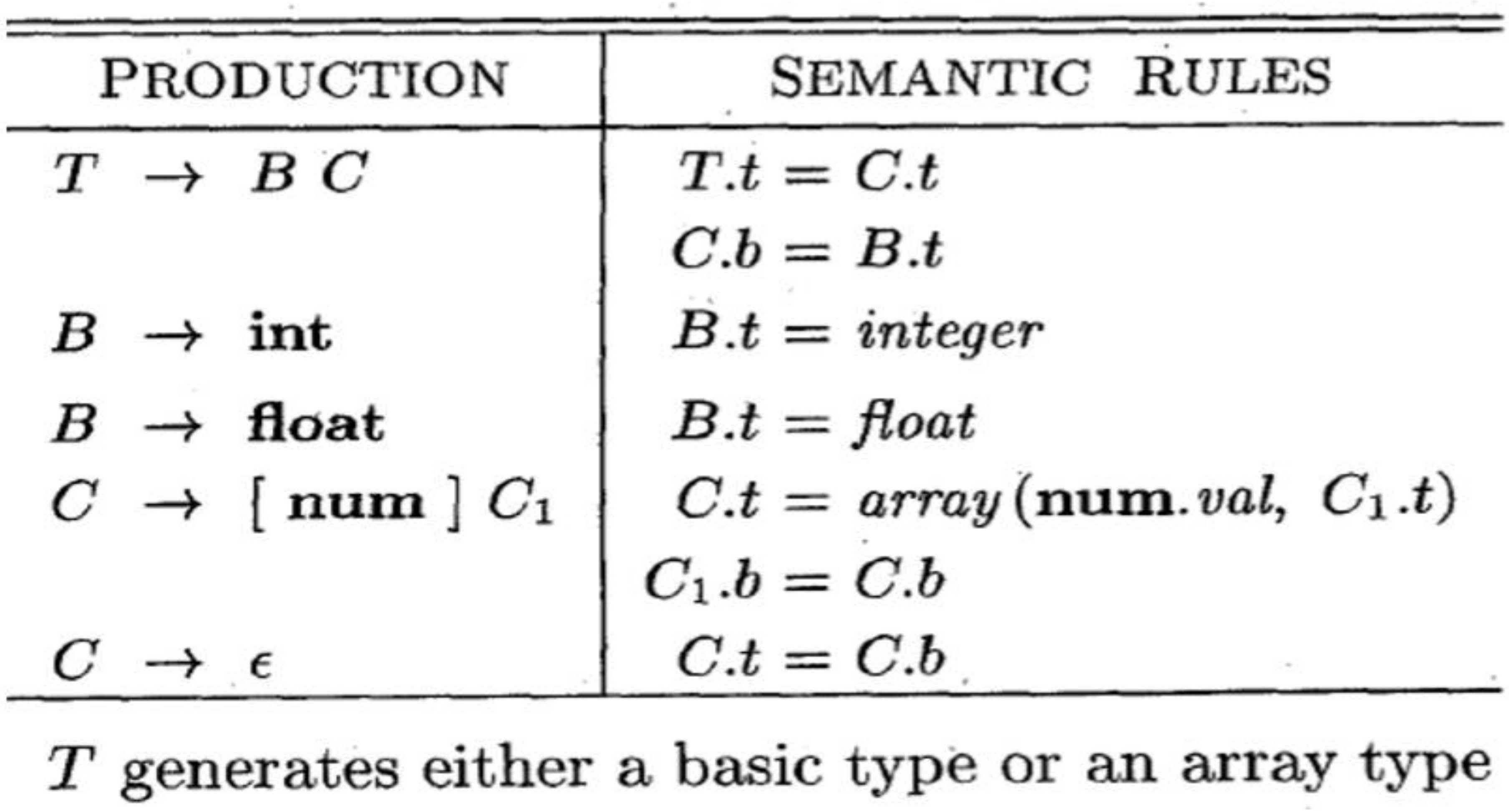
- In C, the type int [2][3] can be read as, “array of 2 arrays of 3 integers”
- The corresponding type expression array(2,array(3,integer)) is represented by the tree in figure



Type expression for **int**[2][3]

- 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

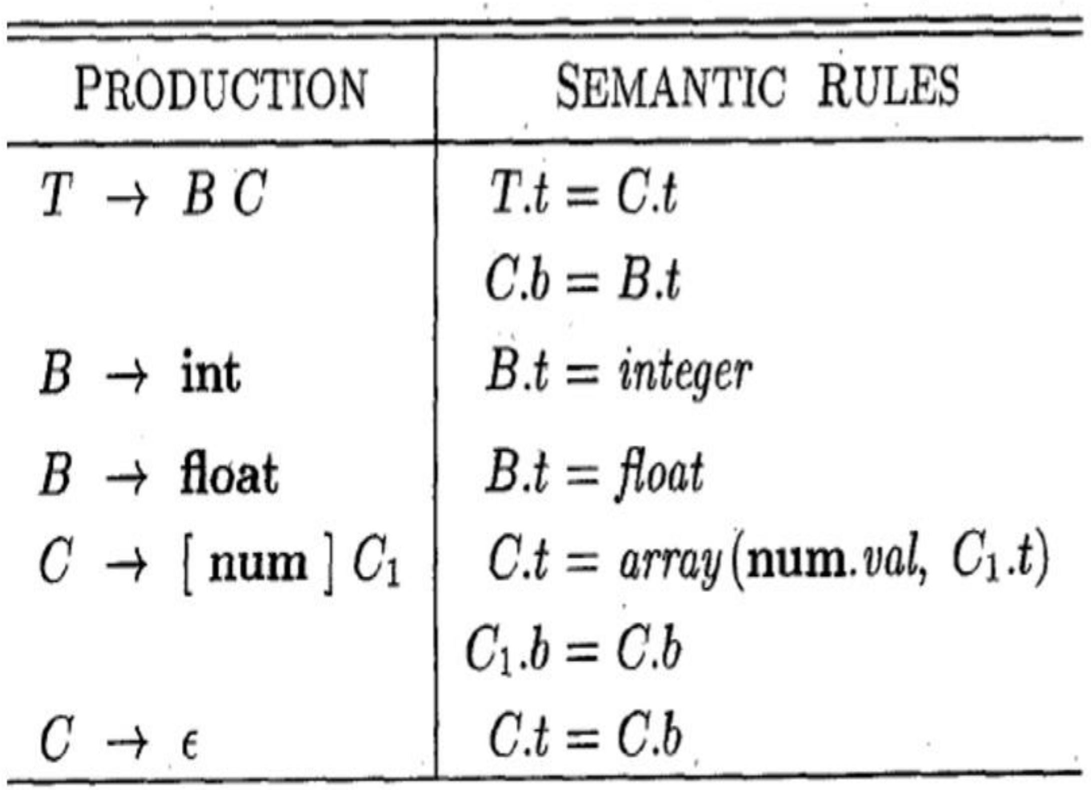
Example 5.13(Cont…)



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

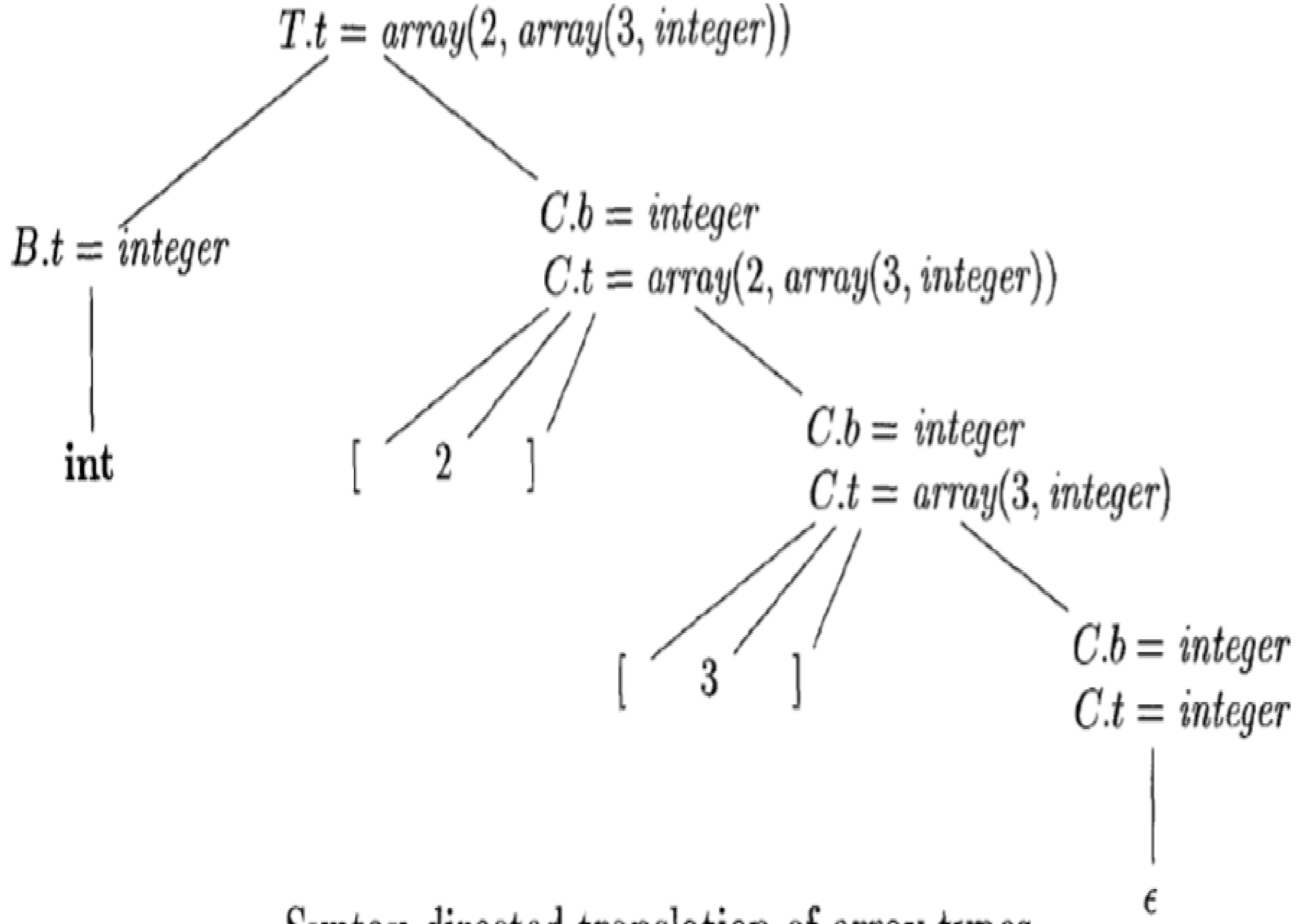
T generates either a basic type or an array type

Example 5.13(Cont…)



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

T generates either a basic type or an array type



$T.t = array(2, array(3, integer))$
$B.t = integer$
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

Syntax-directed translation of array types

Practice Problem

Input String: **3 * 4 * 5**

- Construct for the above expression by following SDD of Example 5.3
  - Parse Tree
  - Annotated Parse Tree
  - Dependency Graph
  - Find out the Topological Sort for the Dependency Graph

Thank you