Syntax-Directed Translation

Dewan Tanvir Ahmed
Assistant Professor, CSE
BUET

Syntax-Directed Translation

1. Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.

2. Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

3. Evaluation of these semantic rules:
   - may generate intermediate codes
   - may put information into the symbol table
   - may perform type checking
   - may issue error messages
   - may perform some other activities
   - in fact, they may perform almost any activities.

4. An attribute may hold almost any thing.
   - a string, a number, a memory location, a complex record.

Syntax-Directed Definitions and Translation Schemes

1. When we associate semantic rules with productions, we use two notations:
   - **Syntax-Directed Definitions**
   - **Translation Schemes**

A. Syntax-Directed Definitions:

- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

B. Translation Schemes:

- indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions

1. A syntax-directed definition is a generalization of a context-free grammar in which:
   - Each grammar symbol is associated with a set of attributes.
   - This set of attributes for a grammar symbol is partitioned into two subsets called
     - **synthesized** and
     - **inherited** attributes of that grammar symbol.
   - Each production rule is associated with a set of semantic rules.

2. *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.

3. This *dependency graph* determines the evaluation order of these semantic rules.

4. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

1. A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

2. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

3. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b=f(c_1,c_2,\ldots,c_n)$$

where f is a function and b can be one of the followings:

➔ b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

➔ b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

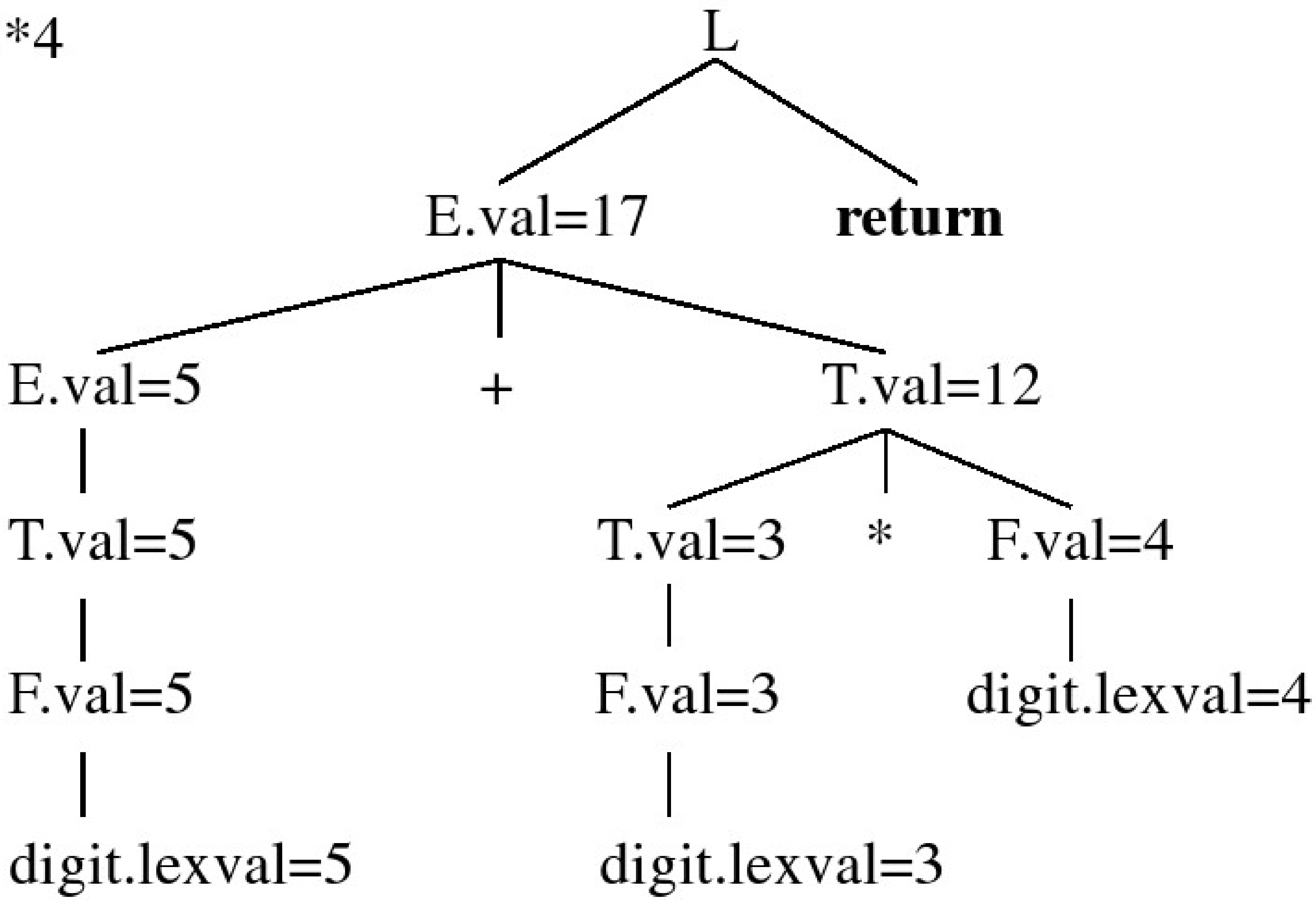
Syntax-Directed Definition -- Example

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

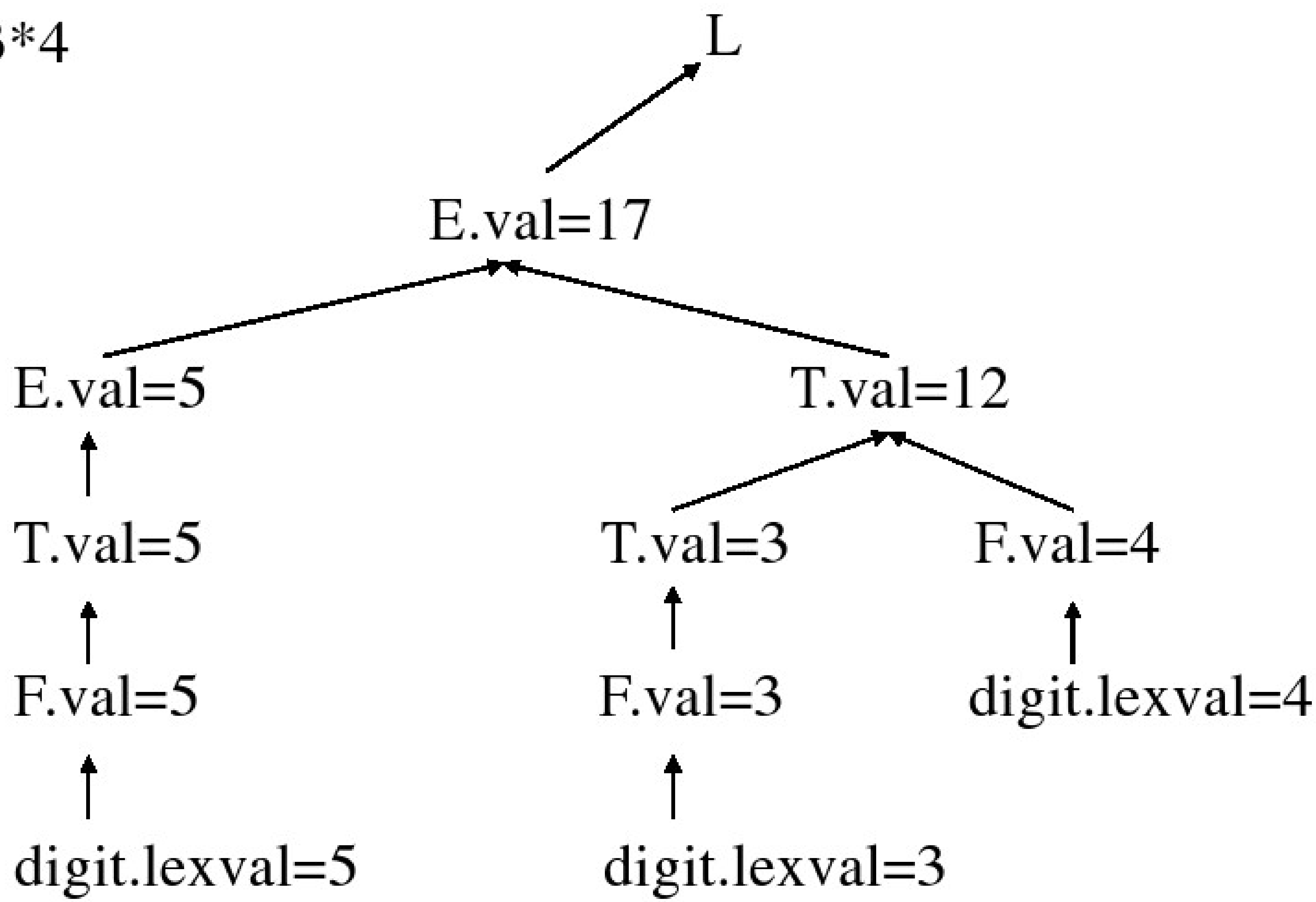
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



Syntax-Directed Definition – Example2

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.loc=newtemp(), E.code = E_1.code ǁ T.code ǁ add E_1.loc,T.loc,E.loc
$E \rightarrow T$	E.loc = T.loc, E.code=T.code
$T \rightarrow T_1 * F$	T.loc=newtemp(), T.code = T_1.code ǁ F.code ǁ mult T_1.loc,F.loc,T.loc
$T \rightarrow F$	T.loc = F.loc, T.code=F.code
$F \rightarrow (E)$	F.loc = E.loc, F.code=E.code
$F \rightarrow$ **id**	F.loc = **id**.name, F.code=""

1. Symbols E, T, and F are associated with synthesized attributes *loc* and *code*.
2. The token **id** has a synthesized attribute *name* (it is assumed that it is evaluated by the lexical analyzer).
3. It is assumed that ǁ is the string concatenation operator.

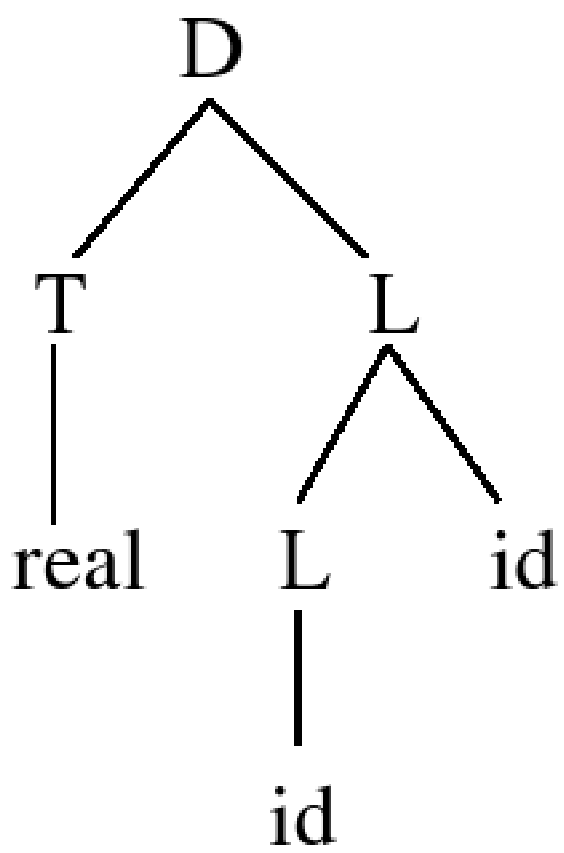
Syntax-Directed Definition – Inherited Attributes

Production	**Semantic Rules**
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$ **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

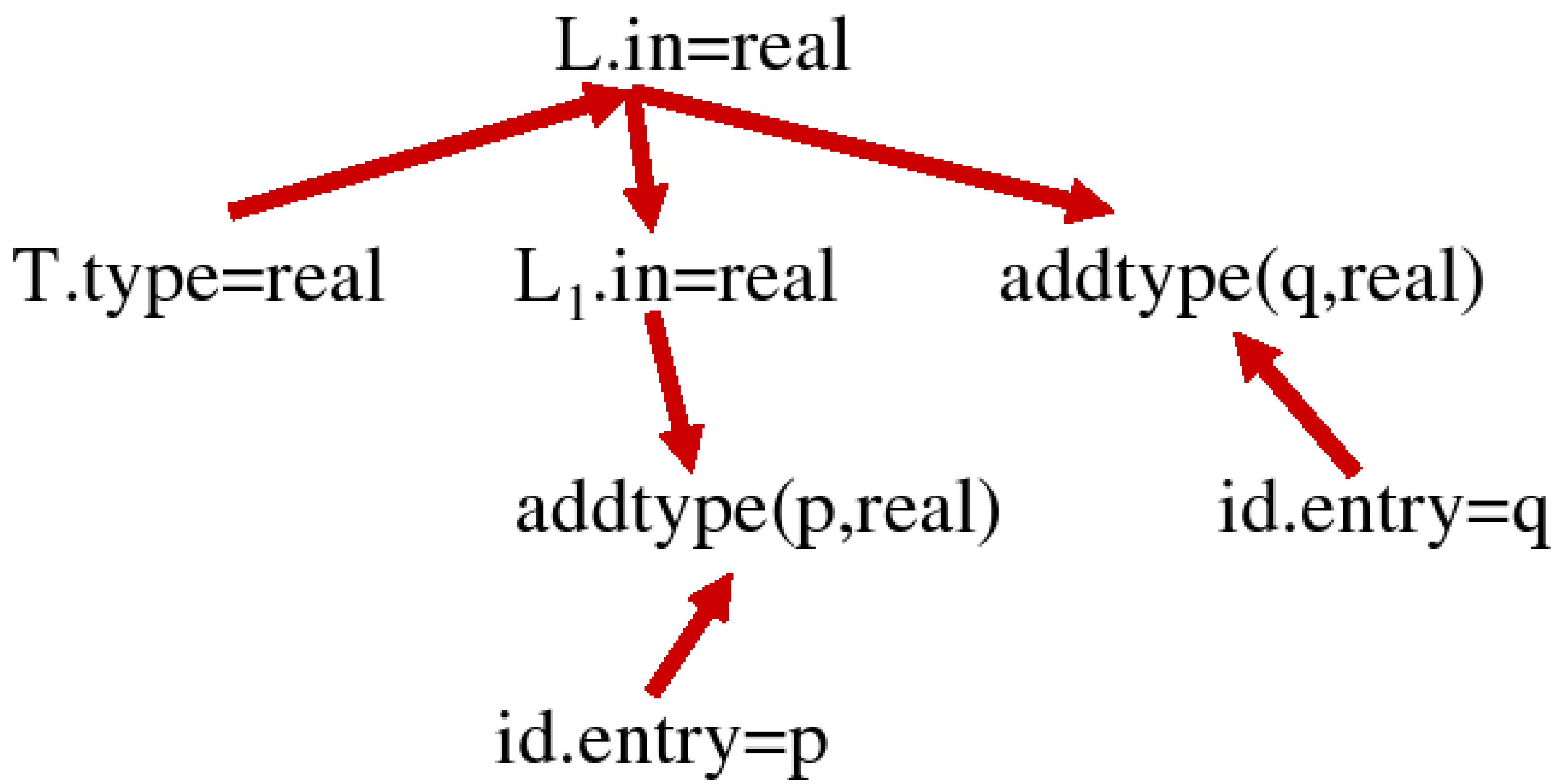
1. Symbol T is associated with a synthesized attribute *type*.
2. Symbol L is associated with an inherited attribute *in*.

A Dependency Graph – Inherited Attributes

Input: `real p q`



parse tree



dependency graph

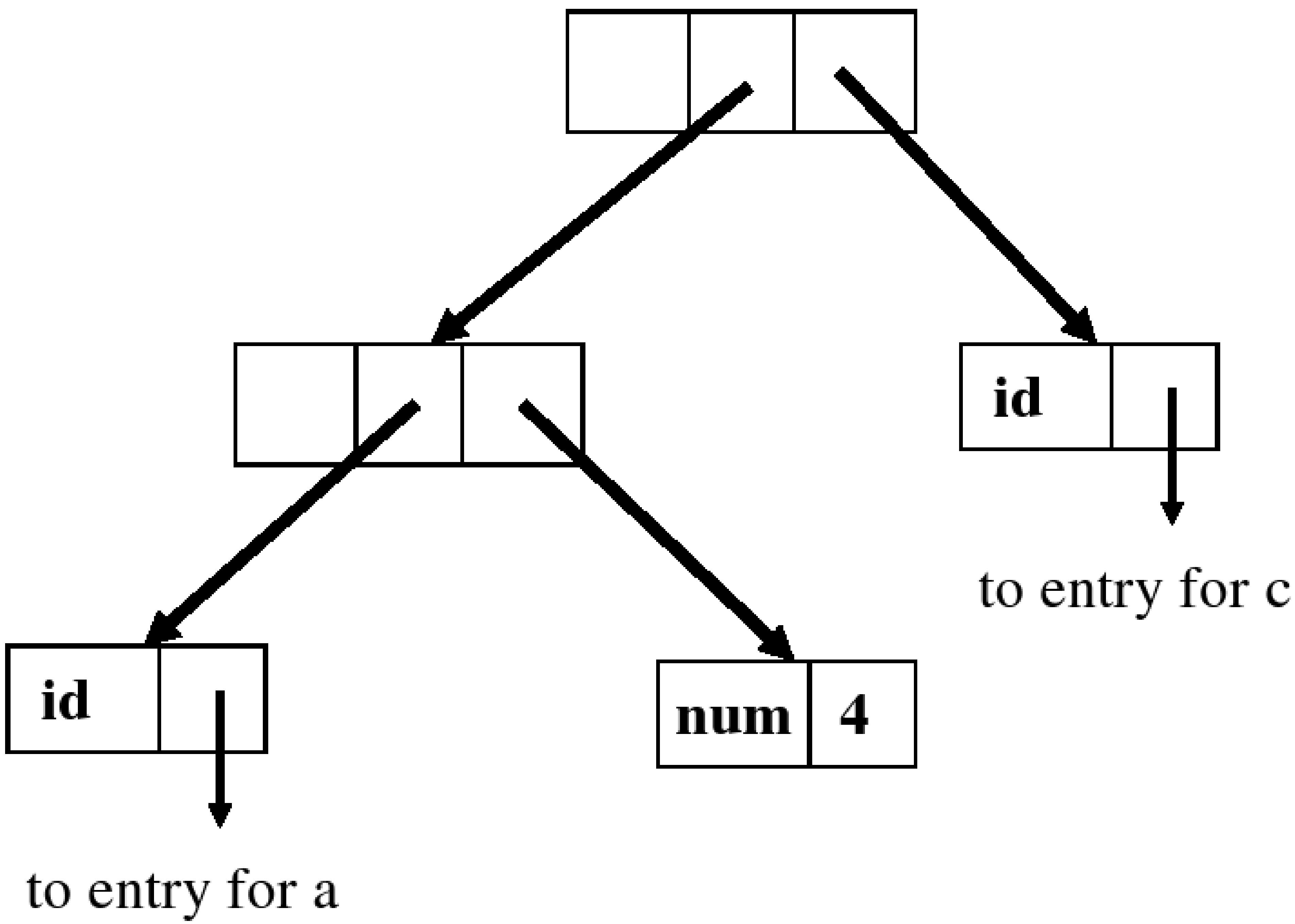
Syntax Trees

1. Decoupling Translation from Parsing-Trees.
2. Syntax-Tree: an intermediate representation of the compiler's input.
3. Example Procedures:
   mknode, *mkleaf*
4. Employment of the synthesized attribute *nptr* (pointer)

PRODUCTION	**SEMANTIC RULE**
$E \rightarrow E_1 + T$	$E.\mathit{nptr} = \mathit{mknode}(\text{“+”}, E_1.\mathit{nptr}, T.\mathit{nptr})$
$E \rightarrow E_1 - T$	$E.\mathit{nptr} = \mathit{mknode}(\text{“-”}, E_1.\mathit{nptr}, T.\mathit{nptr})$
$E \rightarrow T$	$E.\mathit{nptr} = T.\mathit{nptr}$
$T \rightarrow (E)$	$T.\mathit{nptr} = E.\mathit{nptr}$
$T \rightarrow \mathbf{id}$	$T.\mathit{nptr} = \mathit{mkleaf}(\mathbf{id}, \mathbf{id}.\mathit{lexval})$
$T \rightarrow \mathbf{num}$	$T.\mathit{nptr} = \mathit{mkleaf}(\mathbf{num}, \mathbf{num}.\mathit{val})$

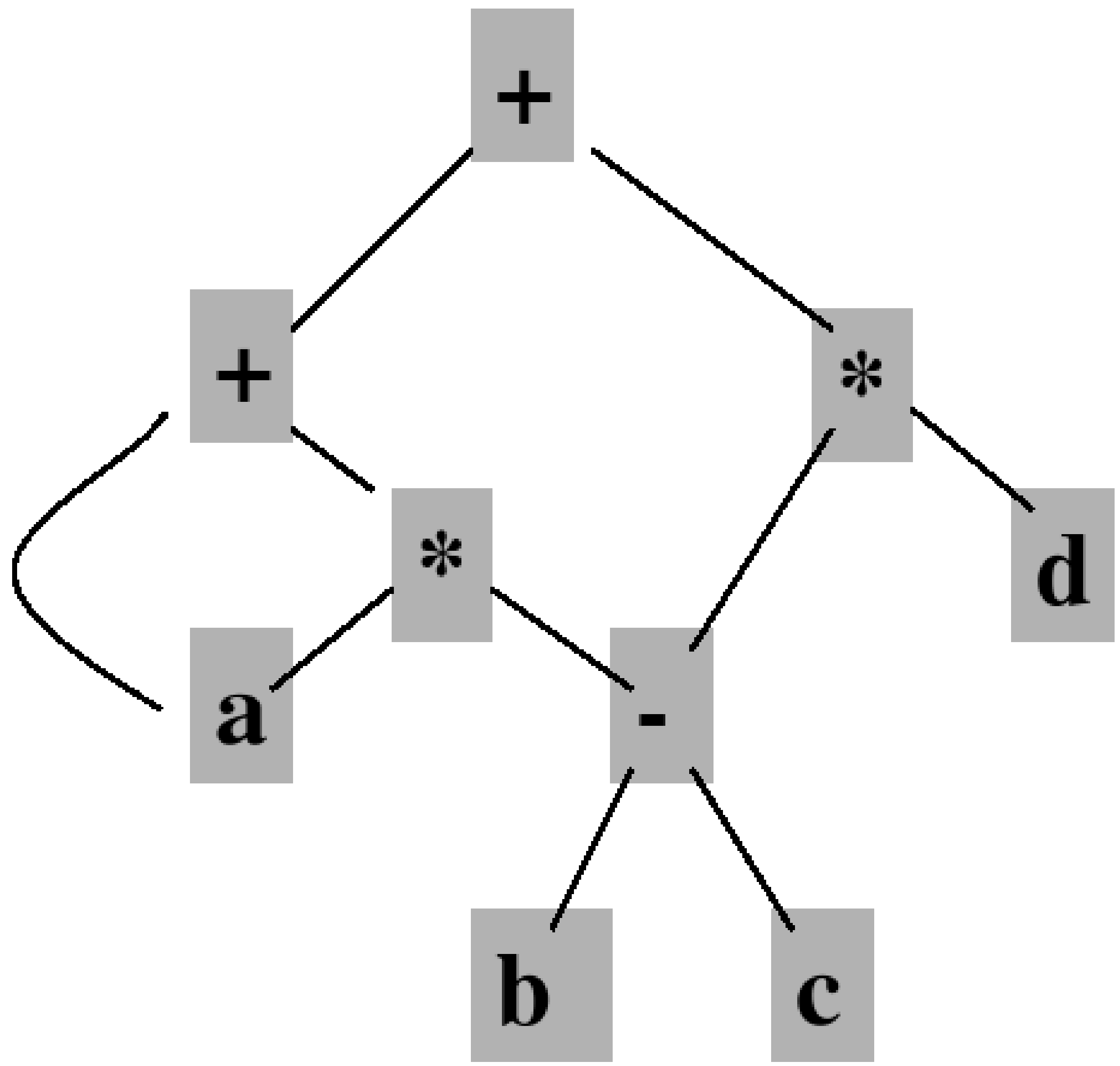
Draw the Syntax Tree

a-4+c



Directed Acyclic Graphs for Expressions

a + a * (b – c) + (b – c) * d



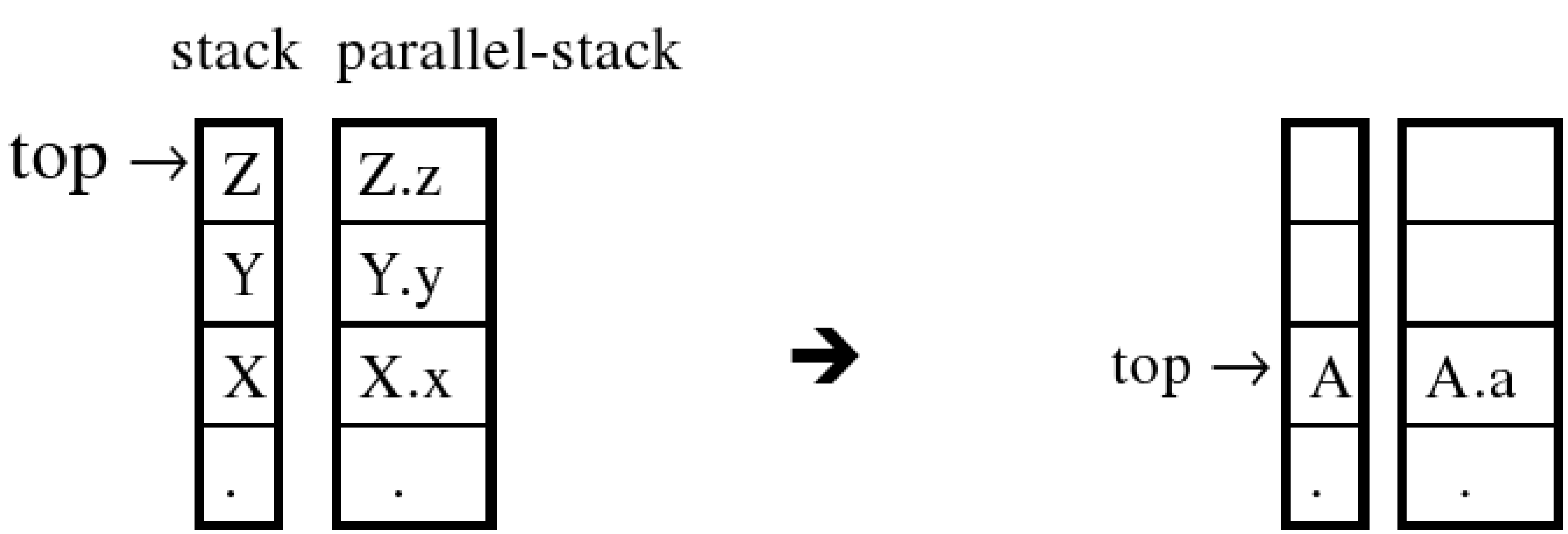
S-Attributed Definitions

1. Syntax-directed definitions are used to specify syntax-directed translations.
2. To create a translator for an arbitrary syntax-directed definition can be difficult.
3. We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
4. We will look at two sub-classes of the syntax-directed definitions:
   - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
   - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
5. To implement S-Attributed Definitions and L-Attributed Definitions we can evaluate semantic rules in a single pass during the parsing.
6. Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

Bottom-Up Evaluation of S-Attributed Definitions

- We put the values of the synthesized attributes of the grammar symbols into a parallel stack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- We evaluate the values of the attributes during reductions.

$A \rightarrow XYZ \qquad A.a=f(X.x,Y.y,Z.z)$ where all attributes are synthesized.



Bottom-Up Eval. of S-Attributed Definitions (cont.)

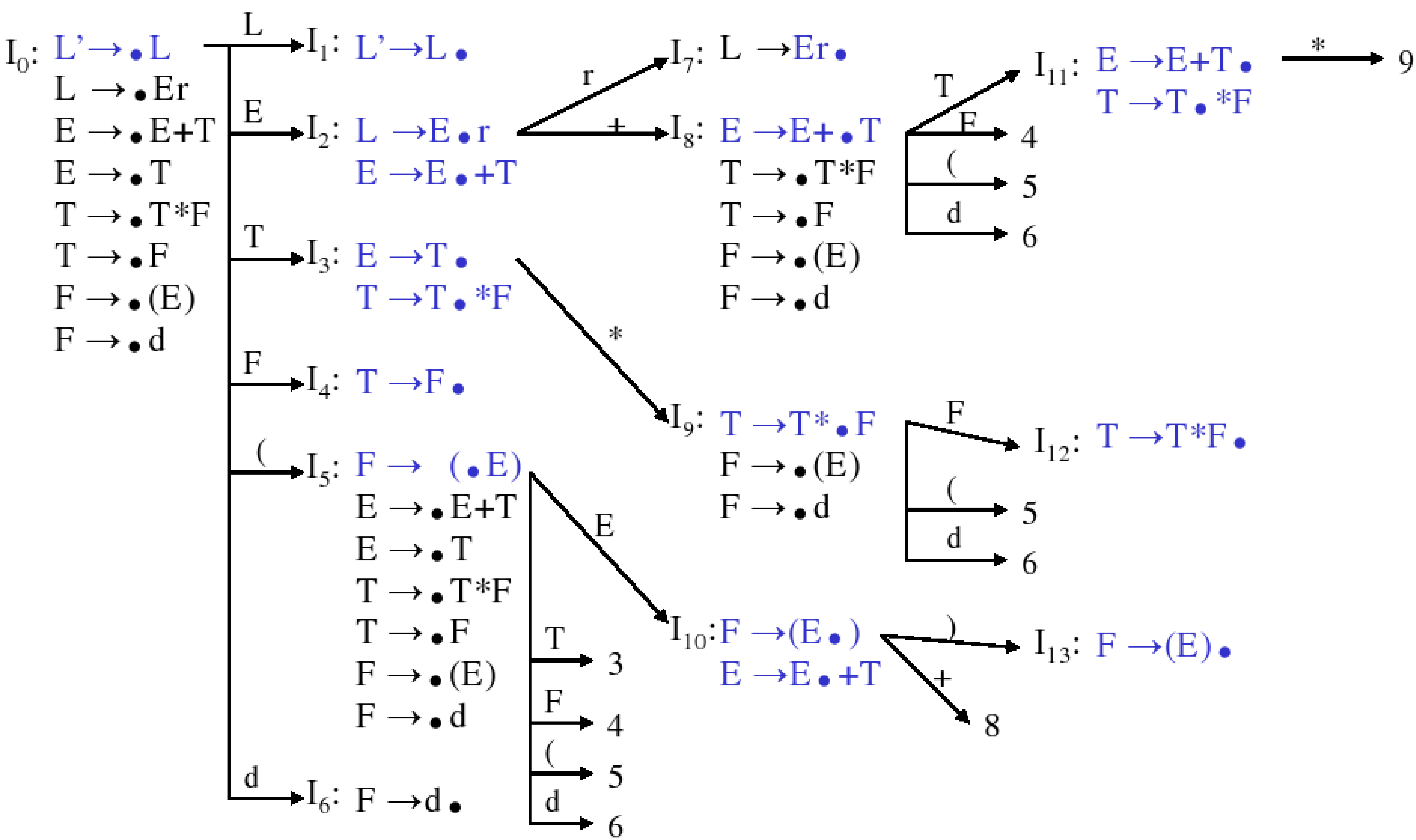
Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(val[top-1])
$E \rightarrow E_1 + T$	val[ntop] = val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntop] = val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] = val[top-1]
$F \rightarrow$ **digit**	

1. At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
2. At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

ntop = top – r + 1

r = no. of symbols in the right side of the production

Canonical LR(0) Collection for The Grammar



Bottom-Up Evaluation -- Example

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

stack	val-stack	input	action	semantic rule
0		5+3*4r	s6	d.lexval(5) into val-stack
0d6	5	+3*4r	F→d	F.val=d.lexval – do nothing
0F4	5	+3*4r	T→F	T.val=F.val – do nothing
0T3	5	+3*4r	E→T	E.val=T.val – do nothing
0E2	5	+3*4r	s8	push empty slot into val-stack
0E2+8	5-	3*4r	s6	d.lexval(3) into val-stack
0E2+8d6	5-3	*4r	F→d	F.val=d.lexval – do nothing
0E2+8F4	5-3	*4r	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4r	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4r	s6	d.lexval(4) into val-stack
0E2+8T11*9d6	5-3-4	r	F→d	F.val=d.lexval – do nothing
0E2+8T11*9F12	5-3-4	r	T→T*F	T.val=T_1.val*F.val
0E2+8T11	5-12	r	E→E+T	E.val=E_1.val*T.val
0E2	17	r	s7	push empty slot into val-stack
0E2r7	17-	$	L→Er	print(17), pop empty slot from val-stack
0L1	17	$	acc	