

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- 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.
- An attribute may hold almost any thing.
 - a string, a number, a memory location, a complex record.



Syntax-Directed Definitions and Translation Schemes

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

- 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.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- 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

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- 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:

$$b = f(c_1, c_2, \dots, c_n) \quad \text{where } f \text{ is a function,}$$

and b can be one of the followings:

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

Attribute Grammar

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

$L \rightarrow E \text{ return}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{digit}$

Semantic Rules

$\text{print}(E.val)$

$E.val = E_1.val + T.val$

$E.val = T.val$

$T.val = T_1.val * F.val$

$T.val = F.val$

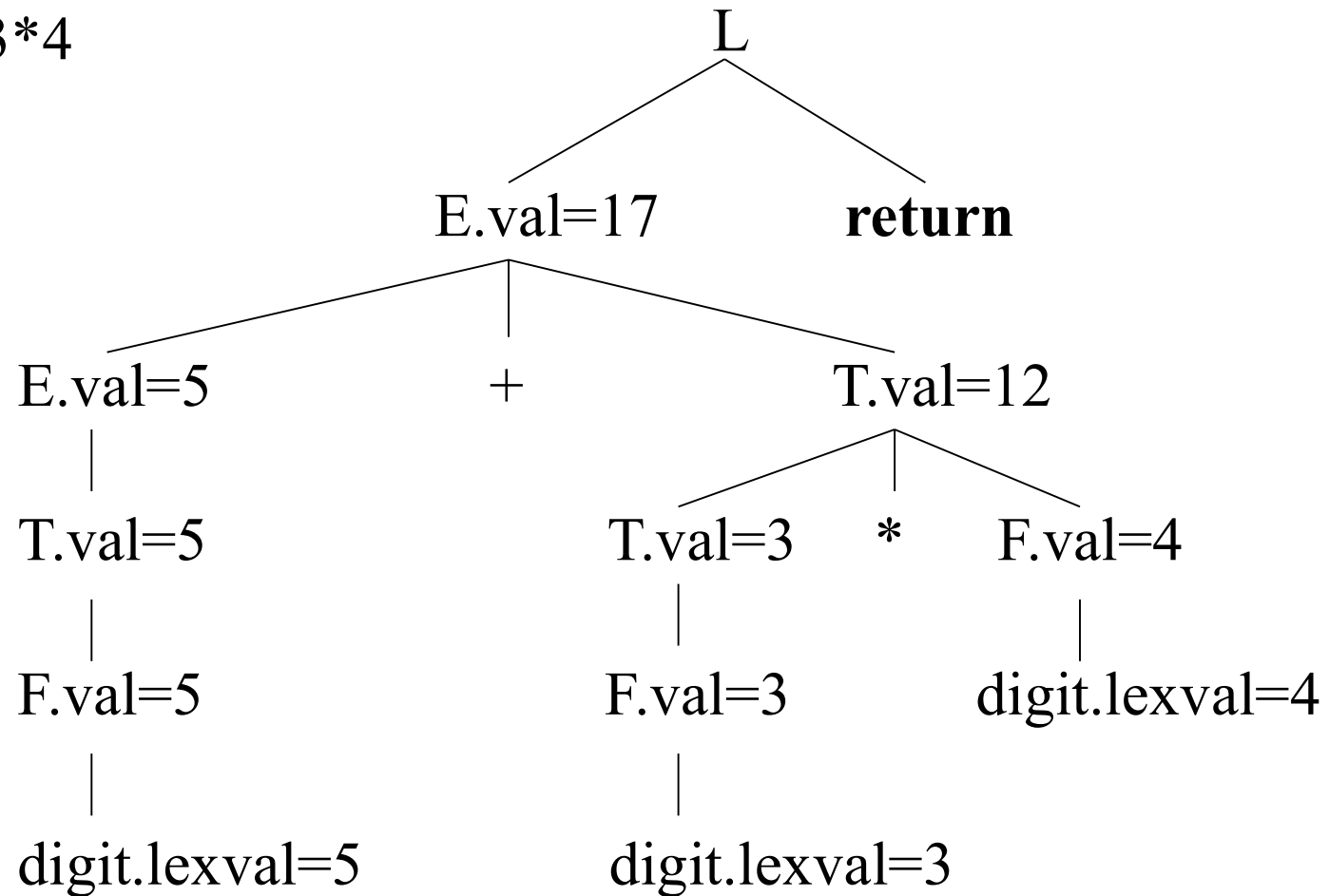
$F.val = E.val$

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

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

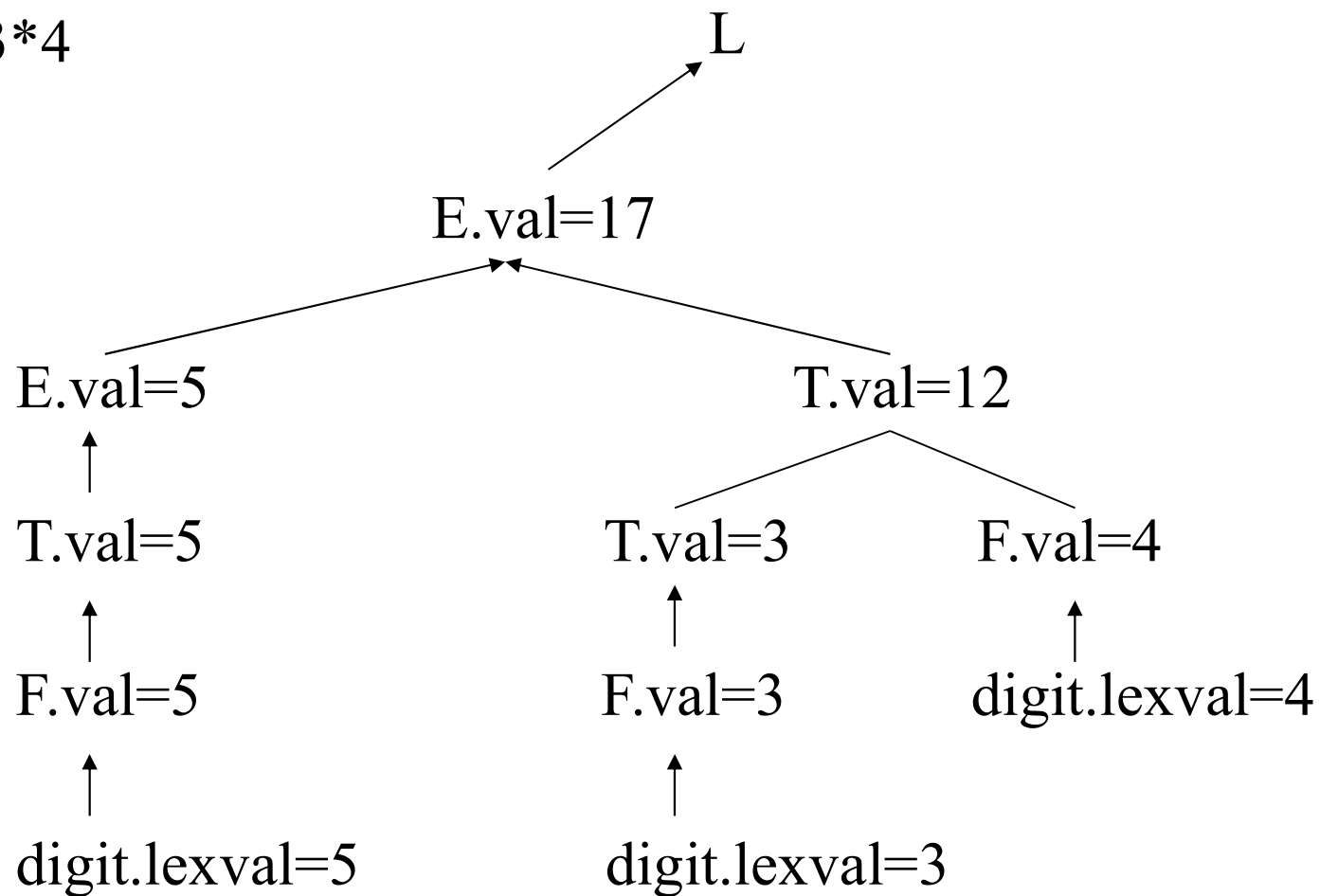
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: $5+3*4$



Syntax-Directed Definition – Example2

<u>Production</u>	<u>Semantic Rules</u>
$E \rightarrow E_1 + T$	$E.loc = \text{newtemp}(), E.code = E_1.code \parallel T.code \parallel \text{add } E_1.loc, T.loc, E.loc$
$E \rightarrow T$	$E.loc = T.loc, E.code = T.code$
$T \rightarrow T_1 * F$	$T.loc = \text{newtemp}(), T.code = T_1.code \parallel F.code \parallel \text{mult } T_1.loc, F.loc, T.loc$
$T \rightarrow F$	$T.loc = F.loc, T.code = F.code$
$F \rightarrow (E)$	$F.loc = E.loc, F.code = E.code$
$F \rightarrow \mathbf{id}$	$F.loc = \mathbf{id.name}, F.code = \text{“”}$

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

Syntax-Directed Definition – Inherited Attributes

Production

Semantic Rules

$D \rightarrow T L$

$L.in = T.type$

$T \rightarrow \mathbf{int}$

$T.type = \text{integer}$

$T \rightarrow \mathbf{real}$

$T.type = \text{real}$

$L \rightarrow L_1 \mathbf{id}$

$L_1.in = L.in, \text{ addtype}(\mathbf{id.entry}, L.in)$

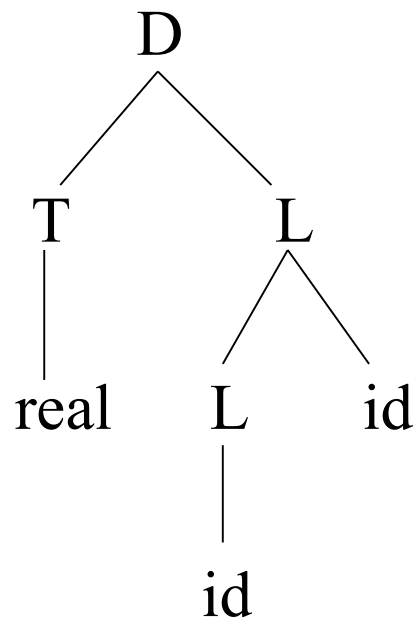
$L \rightarrow \mathbf{id}$

$\text{addtype}(\mathbf{id.entry}, L.in)$

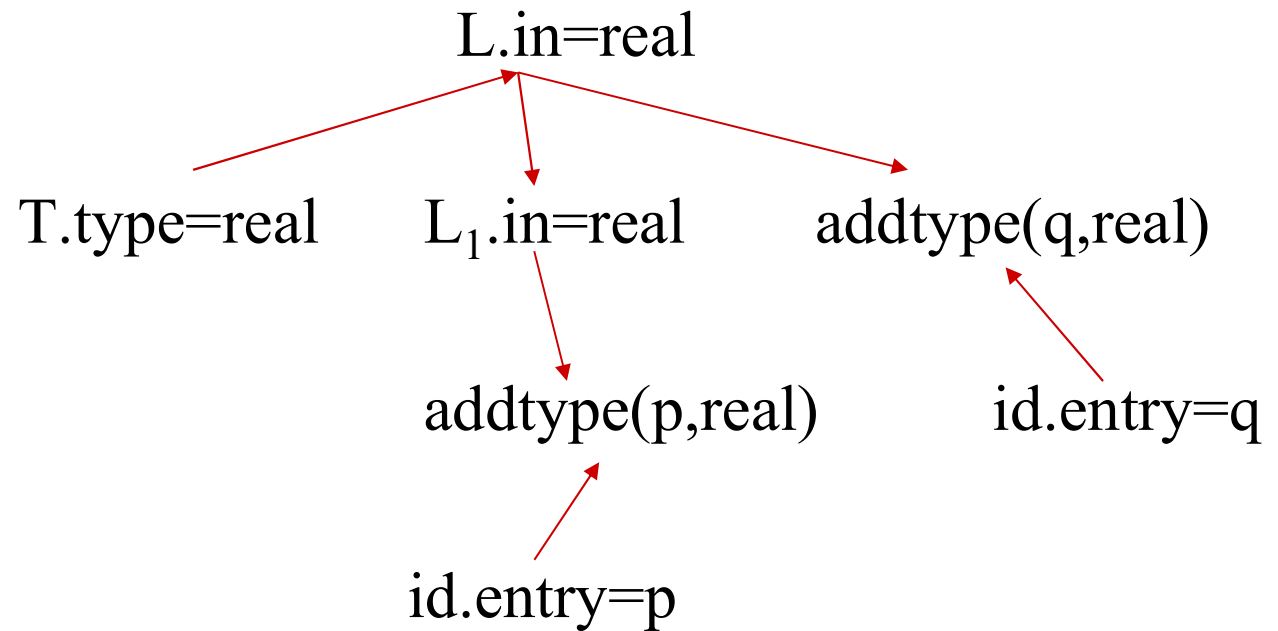
- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

A Dependency Graph – Inherited Attributes

Input: real p q



parse tree



dependency graph

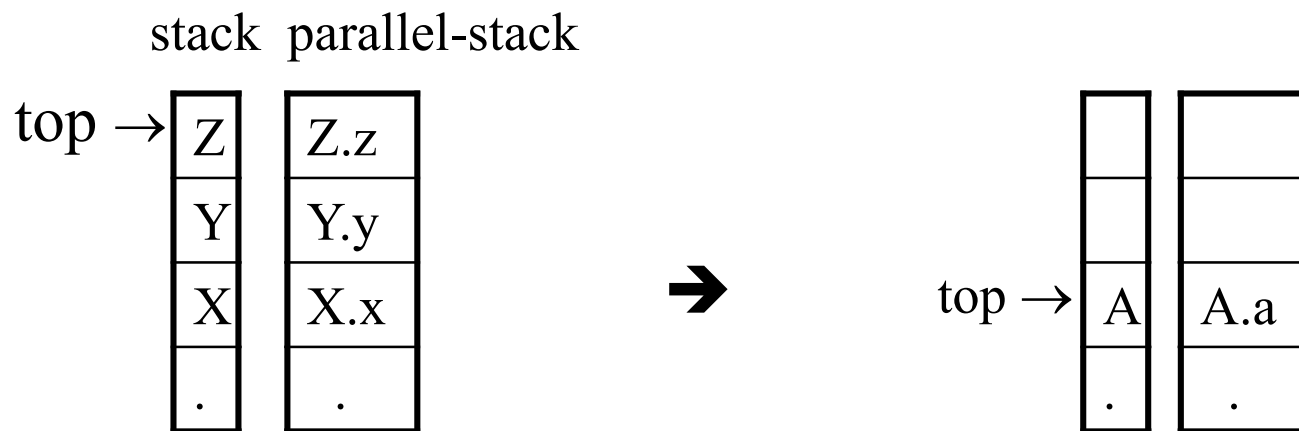
S-Attributed Definitions

- Syntax-directed definitions are used to specify syntax-directed translations.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- 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).
- 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.
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- 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.

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



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

Production

$L \rightarrow E \text{ return}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Semantic Rules

$\text{print}(\text{val}[\text{top}-1])$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] * \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-1]$

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

Bottom-Up Evaluation -- Example

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

<u>stack</u>	<u>val-stack</u>	<u>input</u>	<u>action</u>	<u>semantic rule</u>
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	

L-Attributed Definitions

- S-Attributed Definitions can be efficiently implemented.
- We are looking for a larger (larger than S-Attributed Definitions) subset of syntax-directed definitions which can be efficiently evaluated.

➔ L-Attributed Definitions

- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.
- This means that they can also be evaluated during the parsing.

L-Attributed Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j , where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2 \dots X_n$ depends only on:
 1. The attributes of the symbols X_1, \dots, X_{j-1} to the left of X_j in the production and
 2. the inherited attribute of A
- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

A Definition which is NOT L-Attributed

Productions

$A \rightarrow L M$

$A \rightarrow Q R$

Semantic Rules

$L.in = l(A.i), M.in = m(L.s), A.s = f(M.s)$

$R.in = r(A.in), Q.in = q(R.s), A.s = f(Q.s)$

- This syntax-directed definition is not L-attributed because the semantic rule $Q.in = q(R.s)$ violates the restrictions of L-attributed definitions.
- When $Q.in$ must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of $Q.in$ depends on $R.s$ which will be available after we return from R . So, we are not be able to evaluate the value of $Q.in$ before we enter to Q .