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:

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

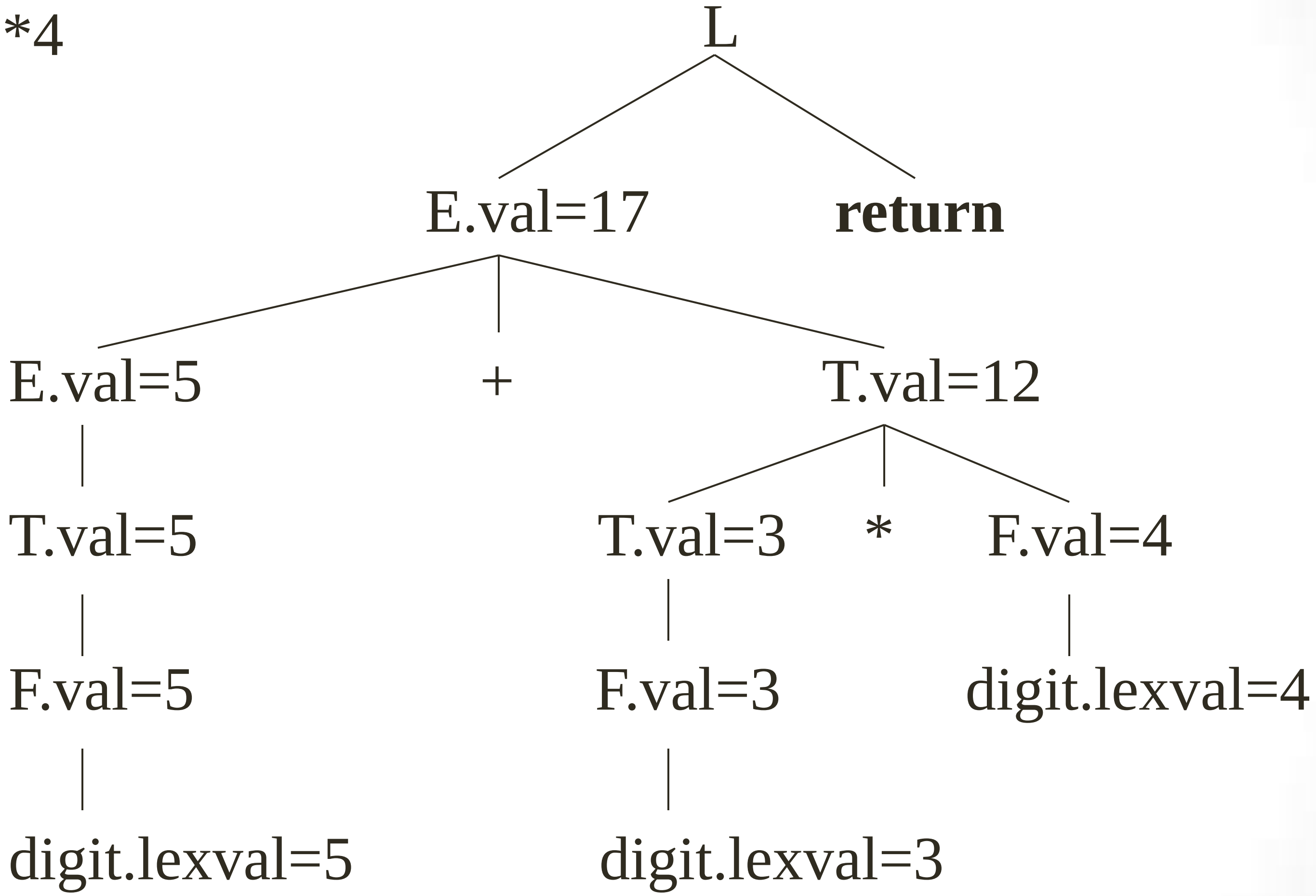
Syntax-Directed Definition -- Example

Production	**Semantic Rules**
L → E **return**	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **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

$$\begin{tikzcd}
 & & L & & & \\
 & E.val=17 \arrow[ur] & & & & \\
E.val=5 \arrow[ur] & & & & T.val=12 \arrow[ulll] & \\
T.val=5 \arrow[u] & & & T.val=3 \arrow[ur, -] & & F.val=4 \arrow[ul, -] \\
F.val=5 \arrow[u] & & & F.val=3 \arrow[u] & & digit.lexval=4 \arrow[u] \\
digit.lexval=5 \arrow[u] & & & digit.lexval=3 \arrow[u] & &
\end{tikzcd}$$

Syntax-Directed Definition – Example2

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.loc=newtemp(), E.code = E_1.code || T.code || 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 || F.code || 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=""

- 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 || is the string concatenation operator.

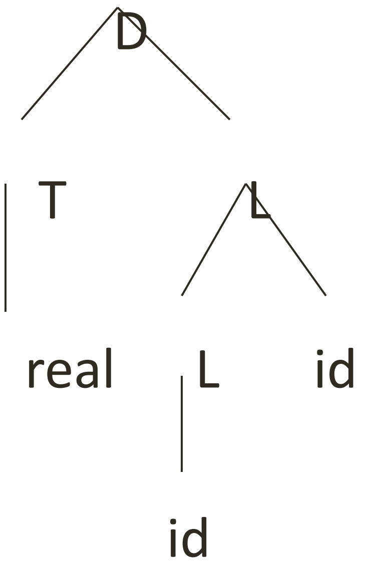
Syntax-Directed Definition – Inherited Attributes

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

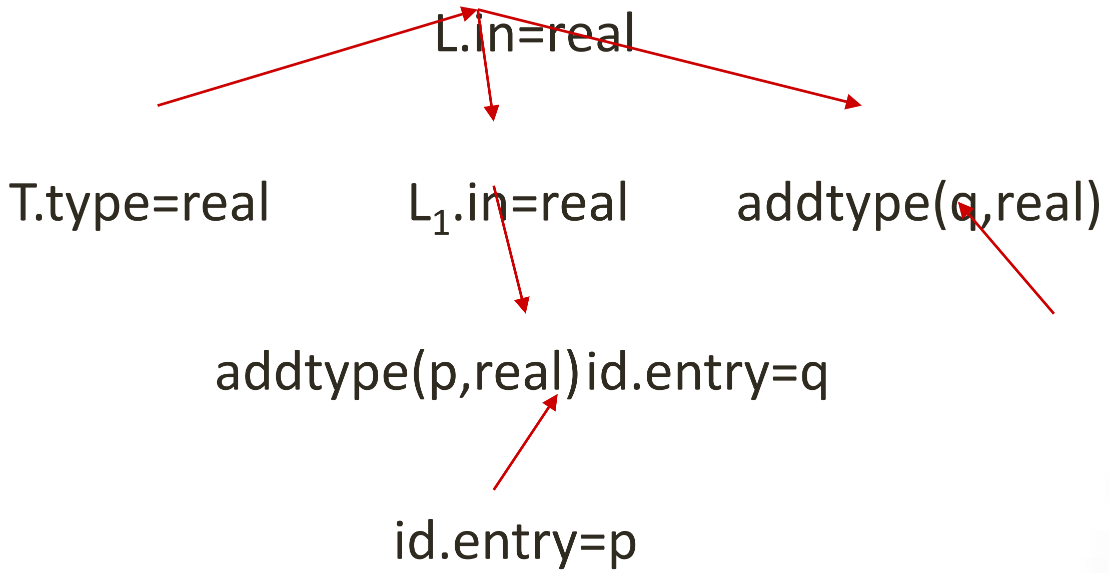
- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in.*

A Dependency Graph – Inherited Attributes

Input: `real p q`



parse tree



dependency graph

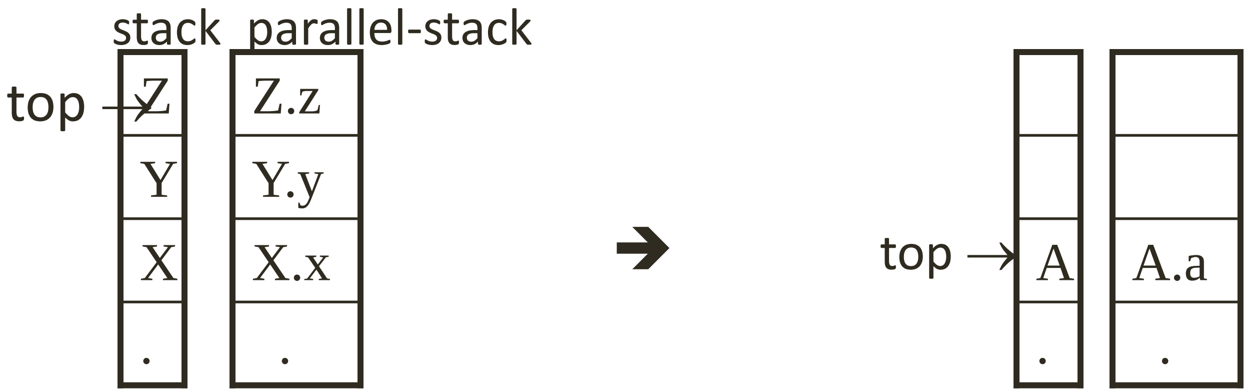
S-Attributed Definitions

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

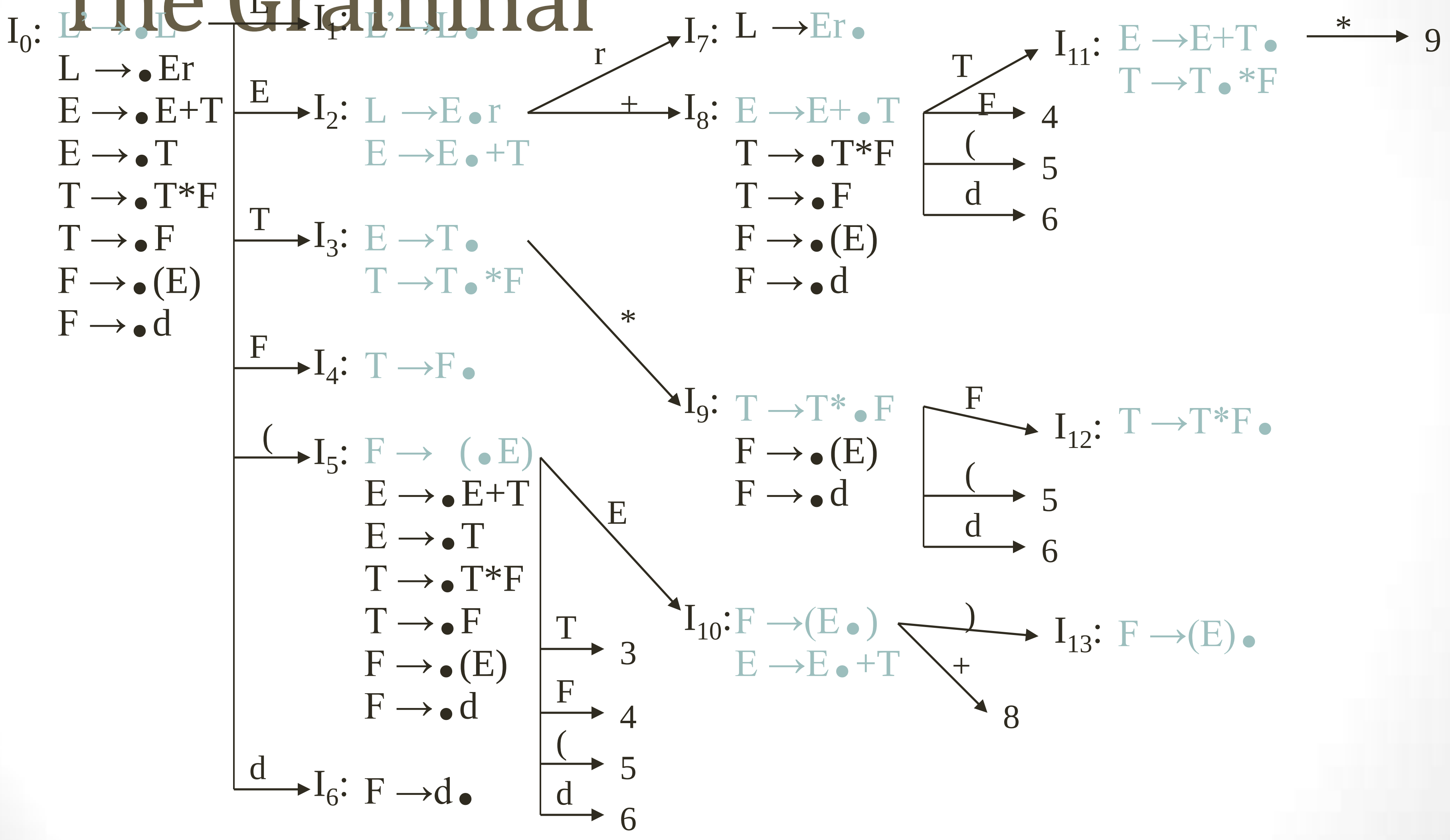


Bottom-Up Eval. of S-Attributed Definitions (cont.)

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

- 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*).

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	

Top-Down Evaluation (of S-Attributed Definitions)

Productions	**Semantic Rules**
$A \rightarrow B$	print(B.n0), print(B.n1)
$B \rightarrow \mathbf{0}\ B_1$	$B.n0=B_1.n0+1$, $B.n1=B_1.n1$
$B \rightarrow \mathbf{1}\ B_1$	$B.n0=B_1.n0$, $B.n1=B_1.n1+1$
$B \rightarrow \varepsilon$	$B.n0=0$, $B.n1=0$

where B has two synthesized attributes (n0 and n1).

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 $1 \le j \le n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on:
  1. The attributes of the symbols $X_1,...,X_{j-1}$ 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	**Semantic Rules**
$A \rightarrow L\ M$	L.in=l(A.i), M.in=m(L.s), A.s=f(M.s)
$A \rightarrow Q\ R$	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.

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).

- A **translation scheme** is a context-free grammar in which:
  - attributes are associated with the grammar symbols and
  - semantic actions enclosed between braces {} are inserted within the right sides of productions.

- *Ex:* A → { ... } X { ... } Y { ... }

Semantic Actions

Translation Schemes

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will be evaluated.

Translation Schemes for S-attributed Definitions

- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production	Semantic Rule	
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	➔ a production of a syntax directed definition

$$\Downarrow$$

$E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$ ➔ the production of the corresponding translation scheme

A Translation Scheme Example

- A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

E → T R

R → + T { print("+") } R_1

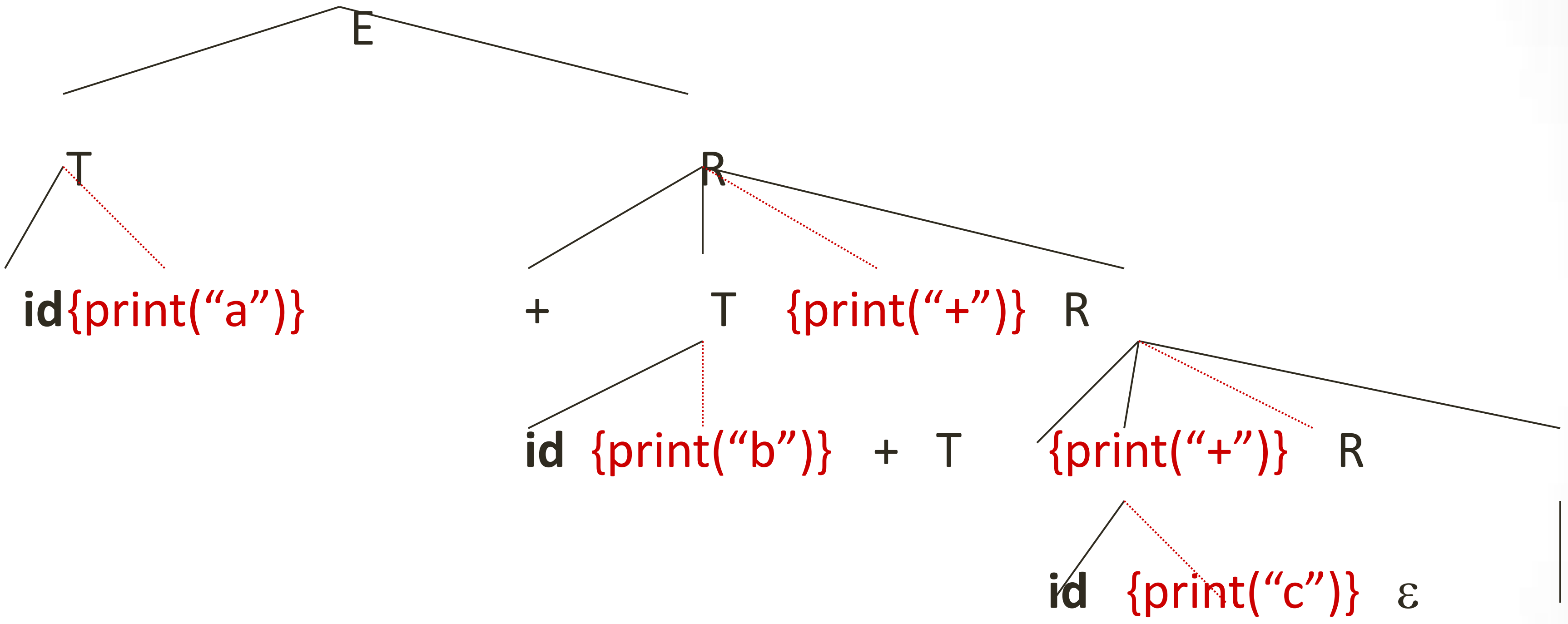
R → ε

T → **id** { print(**id**.name) }

a+b+c ➔ ab+c+

infix expression   postfix expression

A Translation Scheme Example (cont.)



The depth first traversal of the parse tree (executing the semantic actions in that order)

will produce the postfix representation of the infix expression.

A Translation Scheme with Inherited Attributes

D → T **id** { addtype(**id**.entry,T.type), L.in = T.type } L

T → **int** { T.type = integer }

T → **real** { T.type = real }

L → **id** { addtype(**id**.entry,L.in), L_1.in = L.in } L_1

L → ε

- This is a translation scheme for an L-attributed definitions.

Eliminating Left Recursion from Translation Scheme

- A translation scheme with a left recursive grammar.

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }
$E \rightarrow E_1 - T$ { $E.val = E_1.val - T.val$ }
$E \rightarrow T$ { $E.val = T.val$ }
$T \rightarrow T_1 * F$ { $T.val = T_1.val * F.val$ }
$T \rightarrow F$ { $T.val = F.val$ }
$F \rightarrow (E)$ { $F.val = E.val$ }
$F \rightarrow$ **digit** { $F.val =$ **digit**.lexval }

- When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

Eliminating Left Recursion (cont.)

inherited attribute $\searrow$ synthesized attribute $\downarrow$

$E \rightarrow T$ { A.in=T.val } A { E.val=A.syn }

$A \rightarrow + T$ { A_1.in=A.in+T.val } A_1 { A.syn = A_1.syn}

$A \rightarrow - T$ { A_1.in=A.in-T.val } A_1 { A.syn = A_1.syn}

$A \rightarrow \varepsilon$ { A.syn = A.in }

$T \rightarrow F$ { B.in=F.val } B { T.val=B.syn }

$B \rightarrow * F$ { B_1.in=B.in*F.val } B_1 { B.syn = B_1.syn}

$B \rightarrow \varepsilon$ { B.syn = B.in }

$F \rightarrow (E)$ { F.val = E.val }

$F \rightarrow$ **digit** { F.val = **digit**.lexval }

Eliminating Left Recursion (in general)

$A \to A_1\ Y\ \{\ A.a = g(A_1.a, Y.y)\ \}$

$A \to X\ \{\ A.a = f(X.x)\ \}$

a left recursive grammar with synthesized attributes (a,y,x).

$\Downarrow$ eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

$A \to X\ \{\ R.in = f(X.x)\ \}\ R\ \{\ A.a = R.syn\ \}$

$R \to Y\ \{\ R_1.in = g(R.in, Y.y)\ \}\ R_1\ \{\ R.syn = R_1.syn \}$

$R \to \varepsilon\ \ \{\ R.syn = R.in\ \}$