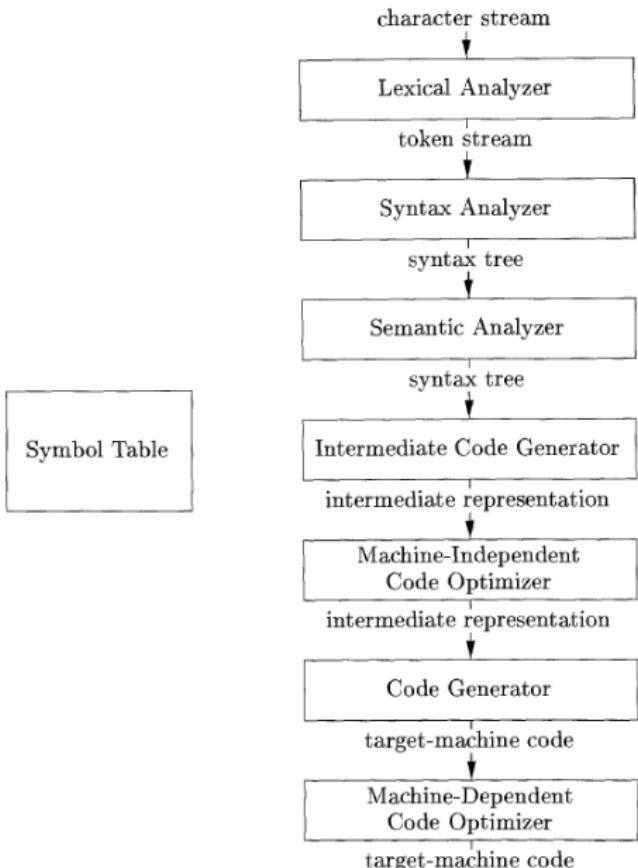
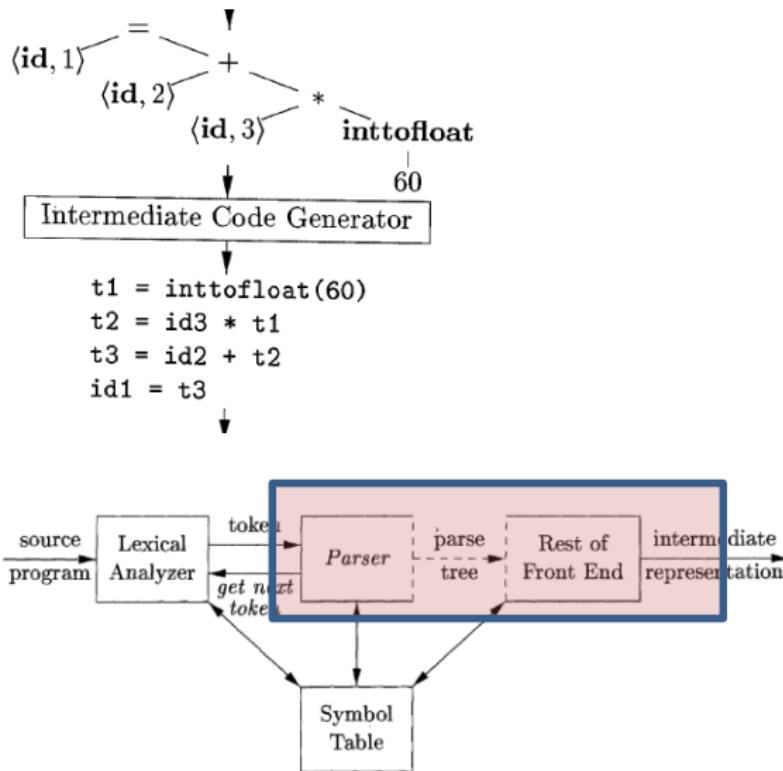


# Syntax-Directed Translation

# The Phases of a Compiler



# Syntax-Directed Translation



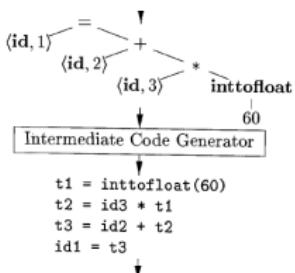
- **Semantic analysis and translation actions can be interlinked with parsing**
- Implemented as a **single module**.

# Syntax-Directed Translation

- Translation of languages guided by context-free grammars.
- Attach *attributes* to the grammar symbol
- Syntax-directed definition specifies the values of attributes
  - By associating semantic rules with the grammar productions

# Syntax-Directed Translation SDD definition

- Syntax-directed definition (SDD) is a context-free grammar together with attributes and rules
  - Attributes are associated with grammar symbols
  - Rules are associated with productions.
- If  $X$  is a grammar symbol and  $a$  is one of its attributes,
  - $X.a$  denotes the value of the attribute  $X$ .
- Attributes may be
  - numbers, types, table references, or strings,
  - Strings may even be code in the intermediate language.



# Attributes

## *Synthesized attribute:*

- *Synthesized attribute* for a **nonterminal A** at a parse-tree node **N** is defined by
- **Semantic rule** associated with the **production at N**.
- 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.**

PRODUCTION

$$E \rightarrow E_1 + T$$

SEMANTIC RULE

$$E.code = E_1.code \parallel T.code \parallel '+'$$

# Attributes

## *Inherited attribute:*

- Inherited attribute for a **nonterminal B** at a parse-tree node N is defined by
- **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**

$$T \rightarrow F T' \quad \mid \quad T'.inh = F.val$$

$$T' \rightarrow * F T'_1 \quad \mid \quad T'_1.inh = T'.inh \times F.val$$

# Attributes

- **Synthesized attribute** at node N to be **defined** in terms of **inherited attribute** values at node **N itself**.

$$T' \rightarrow \epsilon \quad \mid \quad T'.syn = T'.inh$$

- **Do not allow** an **inherited attribute** at node **N** to be defined in terms of attribute values at the **children of node N**
- **Terminals** can have **synthesized attributes**, but not inherited attributes.
- **Attributes for terminals** have **lexical values** that are supplied by the **lexical analyzer**

$$F \rightarrow \text{digit} \quad \mid \quad F.val = \text{digit.lexval}$$

# Example of SDD

Each of the Non-terminals has a **single synthesized attribute**, called *val*

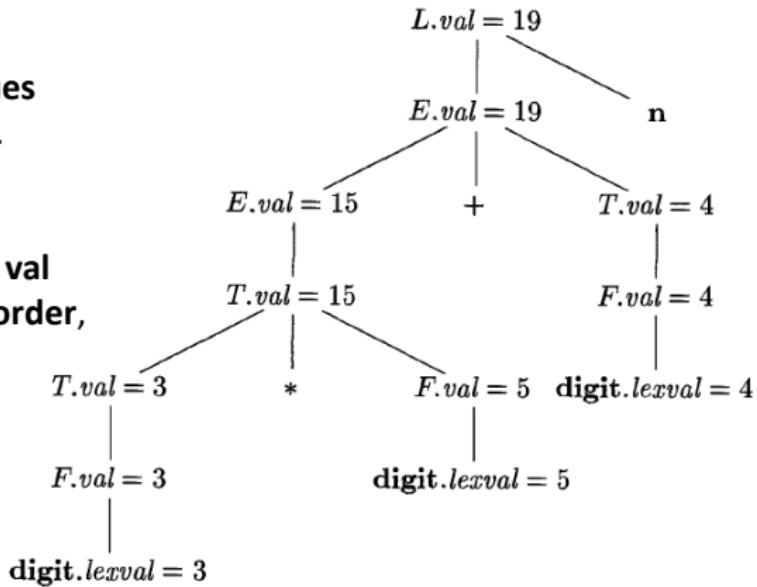
PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow ( E )$	$F.val = E.val$
7)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

# Annotated parse tree.

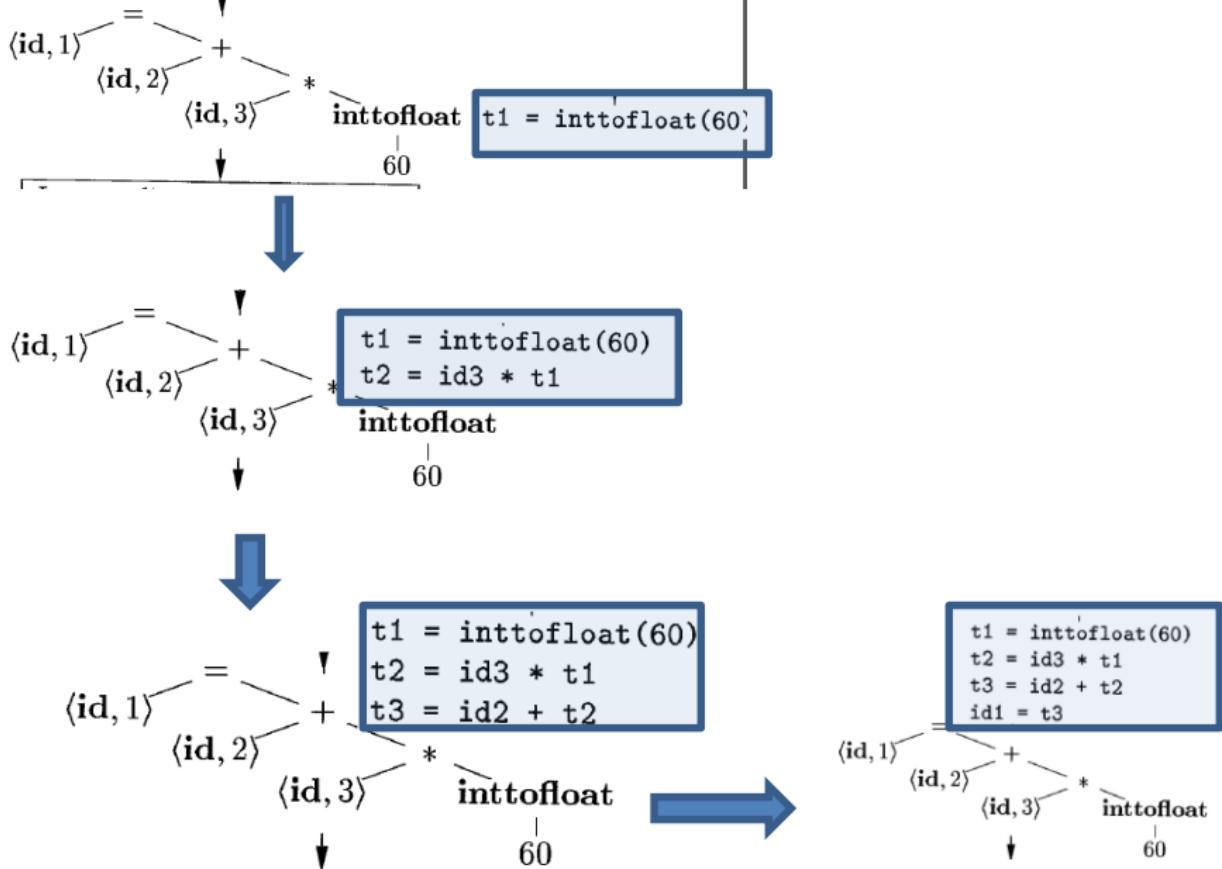
A **parse tree**, showing the **value(s)** of its **attribute(s)** is called an *annotated parse tree*.

Input string: **3 \* 5 + 4 n**

- We show the resulting **values** associated with **each node**.
- Each of the nodes for the nonterminals has **attribute val** computed in a **bottom-up order**,

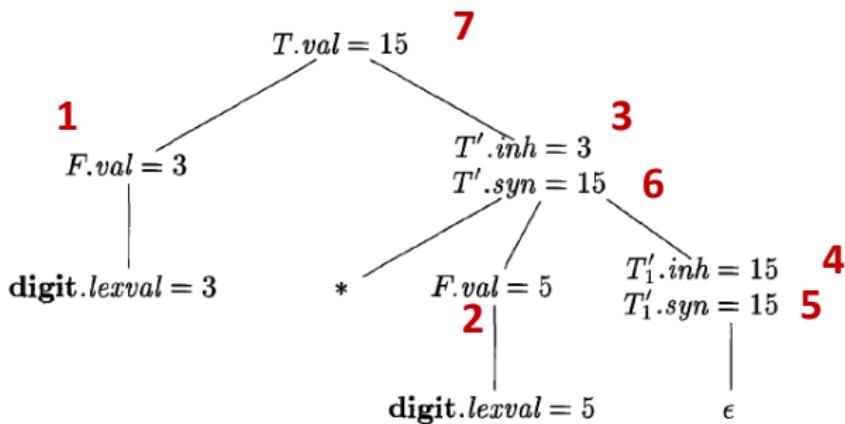


# Annotation and Evaluation of parse tree



# Annotated parse tree.

PRODUCTION	SEMANTIC RULES	
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$	<b>val</b> and <b>syn</b> : Synthesized <b>inh</b> : Inherited
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) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$	<b>Annotated parse tree for 3 * 5</b>



# Evaluation Orders of SDD

- "Dependency graphs" are a useful tool for determining an **evaluation order** for the **attribute** instances in a given parse tree.
  - Depicts the **flow of information** among the attribute instances in a particular parse tree
  - **Directed edges**
- For a **node A in parse tree** -> **node A in dependency graph**

A has a **synthesized** attribute b

**Production**

A->...X..

**Semantic Rule**

A.b=f(.., X.c, ..)

- **Edge** from X.c to A.b

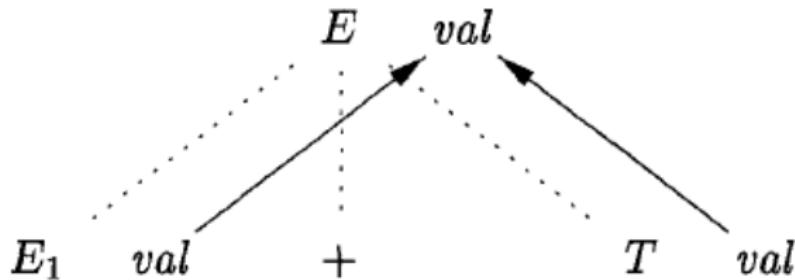
- Edge from **child attribute** to **parent attribute**

PRODUCTION

$$E \rightarrow E_1 + T$$

SEMANTIC RULE

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



# Evaluation Orders of SDD

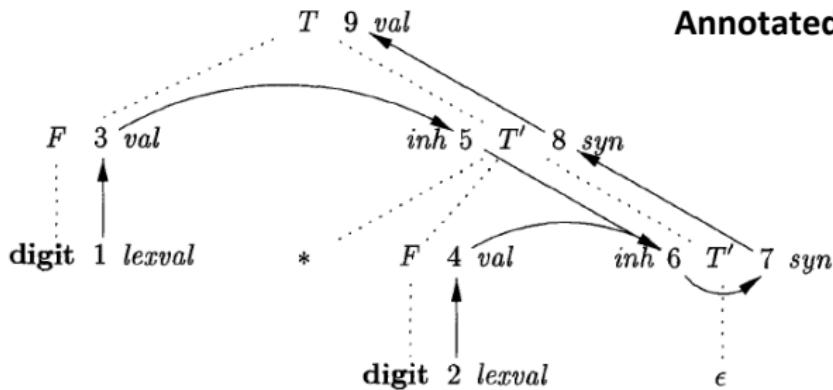
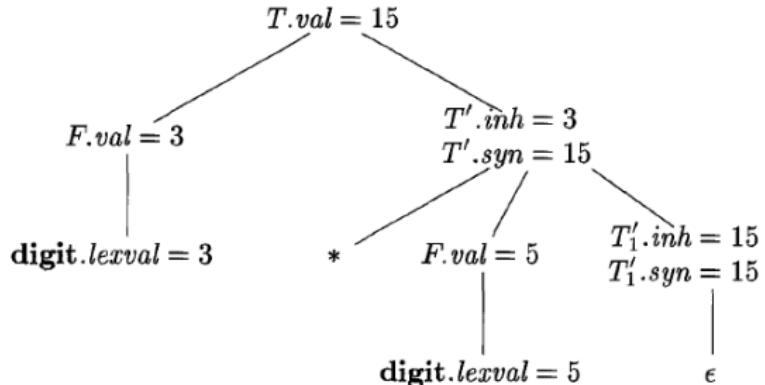
- "Dependency graphs" are a useful tool for determining an **evaluation order** for the attribute instances in a given parse tree.
  - Depicts the flow of information among the attribute instances in a particular parse tree
  - Directed edges
- For a **node A in parse tree** -> **node A in dependency graph**

B has an inherited attribute c

Production	Semantic Rule
A->...B..X..	$B.c=f(.., X.a, ..)$

- **Edge from X.a to B.c**
  - Edge **from attribute a of X** (parent or sibling of B) **to attribute c of B** (body of the production)

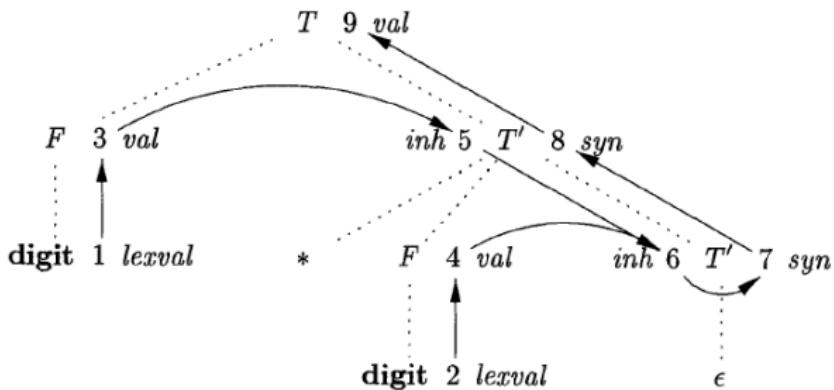
PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$



# Ordering the Evaluation of Attributes

- The **dependency graph** characterizes the possible **evaluation 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**,
  - Attribute corresponding to **M must be evaluated before** the attribute of **N**.
- If there is an edge of the dependency graph from **N<sub>i</sub> to N<sub>j</sub>, such that i < j**
  - the only allowable orders of evaluation are those sequences of nodes **N<sub>1</sub>, N<sub>2</sub>,..., N<sub>k</sub>**
- Embeds a directed graph into a linear order, and is called a **topological sort** of the graph

# Topological Sort- Ordering the Evaluation



- One **topological sort** is the order in which the nodes have already been numbered: 1,2,... ,9.
- There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9.

# Ordering the Evaluation – Cycles

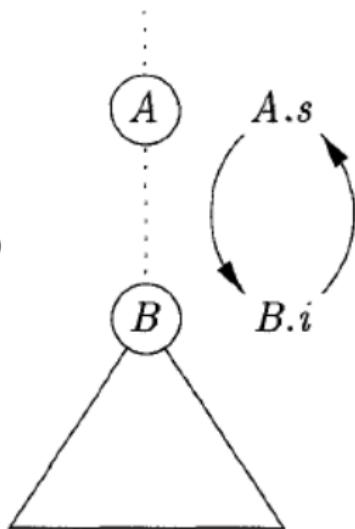
PRODUCTION

$$A \rightarrow B$$

SEMANTIC RULES

$$\begin{aligned} A.s &= B.i; \\ B.i &= A.s + 1 \end{aligned}$$

These rules are circular; it is impossible to evaluate either  $A.s$  or  $B.i$



# Classes of SDD

- (a) S-Attributed Definitions
- (b) L-Attributed Definitions

Guarantee an evaluation order

# S-Attributed SDD

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

PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
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) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

# L-Attributed SDD

- The idea behind L-attributed SDD 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")
1. Synthesized, or
  2. Inherited, but with the rules limited as follows. Suppose that there is a production  $A \rightarrow X_1 X_2 \cdots X_n$ , 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  $X_1, X_2, \dots, X_{i-1}$  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  $X_i$ .

# L-Attributed SDD

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

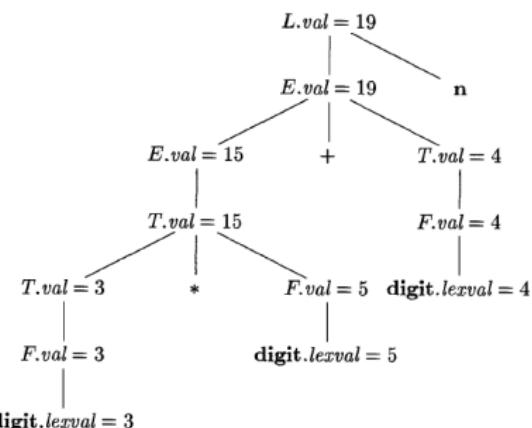
PRODUCTION      SEMANTIC RULES  
 $A \rightarrow B C$        $A.s = B.b;$   
                           $B.i = f(C.c, A.s)$

# Side Effects

- Print a result,
- Enter the type of an identifier into a symbol table.

PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
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) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

PRODUCTION	SEMANTIC RULE
1) $L \rightarrow E \text{ n}$	$print(E.val)$



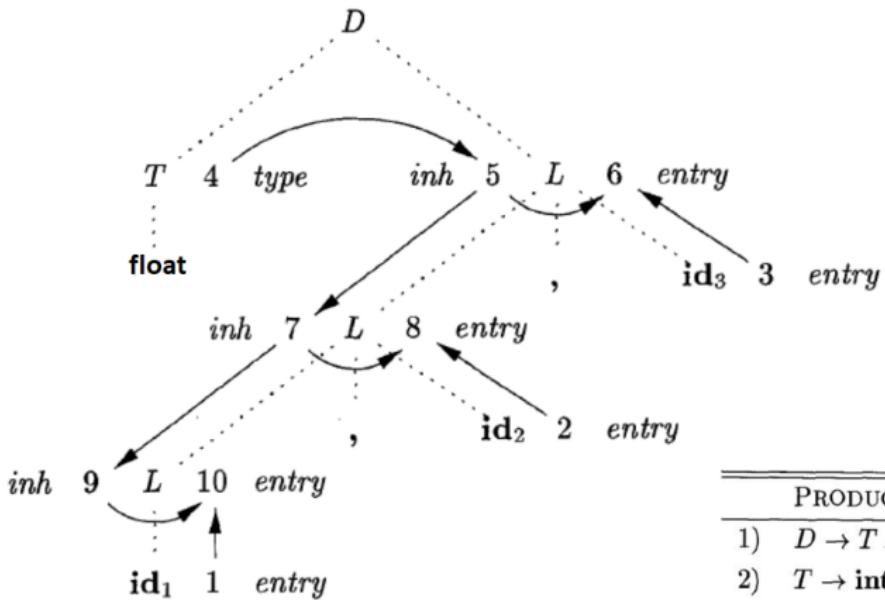
# Side Effects – examples

- The SDD 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.

PRODUCTION	SEMANTIC RULES
1) $D \rightarrow T\ L$	$L.inh = T.type$ ← The type is passed to the attribute L.inh
2) $T \rightarrow \text{int}$	$T.type = \text{integer}$
3) $T \rightarrow \text{float}$	$T.type = \text{float}$ ← Evaluate the synthesized attribute T.type, giving it the appropriate value, integer or float.
4) $L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$ ← Passes L.inh down the parse tree $\text{addType}(\text{id}.entry, L.inh)$ Function addType() properly installs the type L.inh as the type of the identifier.
5) $L \rightarrow \text{id}$	$\text{addType}(\text{id}.entry, L.inh)$ type L.inh as the type of the identifier.

# Side Effects

```
float id1, id2, id3
```



PRODUCTION	SEMANTIC RULES
1) $D \rightarrow T L$	$L.inh = T.type$
2) $T \rightarrow \text{int}$	$T.type = \text{integer}$
3) $T \rightarrow \text{float}$	$T.type = \text{float}$
4) $L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$ $\text{addType}(\text{id}.entry, L.inh)$
5) $L \rightarrow \text{id}$	$\text{addType}(\text{id}.entry, L.inh)$

# Declaration statement

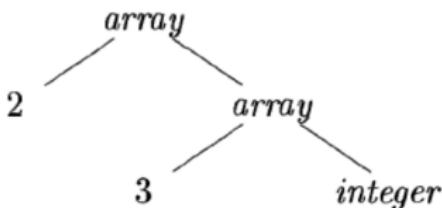
Representing data types: **Type Expressions**

Types have structure, which we shall represent using type expressions.

- A **type expression** is either a **basic type** (*boolean, char, integer, float, and void*)  
or
- is formed by **applying an operator** called a **type constructor** to a type expression.
- A **type expression** can be formed by applying the **array type constructor** to a **number and a type expression**.

# Declaration statement

- The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each"
- Can be represented as a **type expression** array(2, array(3, integer)).
- This type is represented by the tree.



- The **operator array** takes **two parameters**, a number and a type.
  - Here the **type expression** can be formed by applying the **array type constructor** to a number and a type expression.

# Declaration statement

## Example SDD

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



Type Expressions

- 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 C derives  $\epsilon$ .
- Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

# Declaration statement

## Example SDD

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



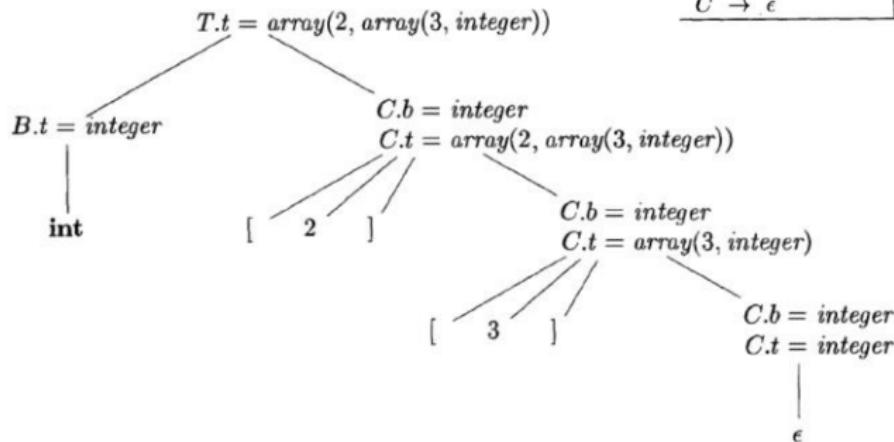
Type Expressions

- The nonterminals  $B$  and  $T$  have a synthesized attribute  $t$  representing a type.
- The nonterminal  $C$  has two attributes: an inherited attribute  $b$  and a synthesized attribute  $t$ .

# Declaration statement

## Example SDD

input string int [2][3]



- The nonterminal  $C$  has two attributes: an inherited attribute  $b$  and a synthesized attribute  $t$ .
- The inherited  $b$  attributes pass a basic type down the tree, and the synthesized  $t$  attributes accumulate the result.

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

# Application of SDD – Syntax tree construction

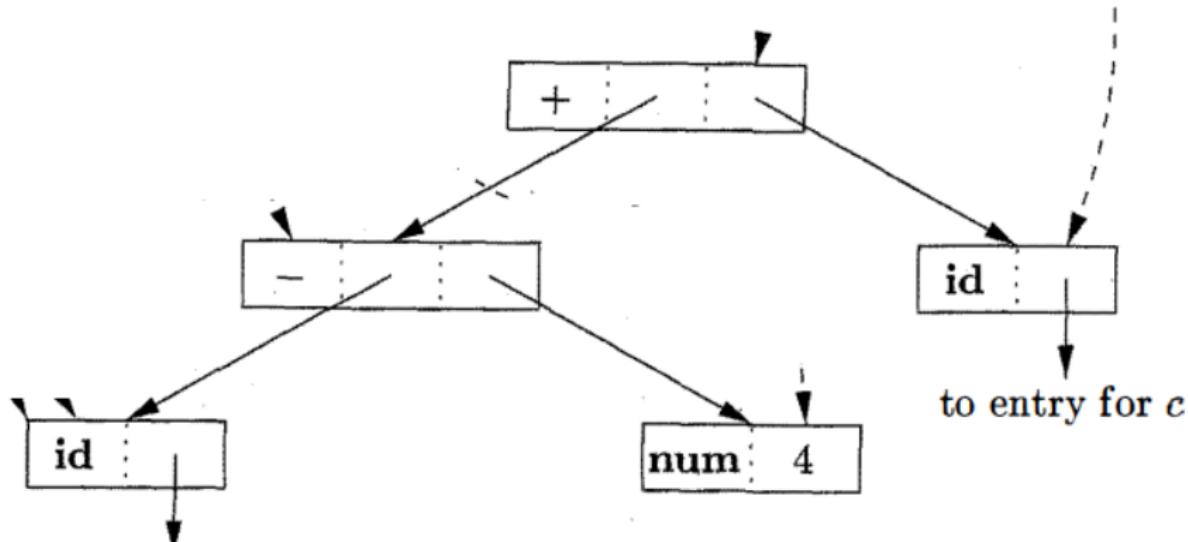
- **Each node** in a syntax tree represents a **construct**; the children of the node represent the meaningful components of the construct.
- A syntax-tree node representing an expression  $E1 + E2$  has label  $+$  and two children representing the subexpressions  $E1$  and  $E2$

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

# Application of SDD – Syntax tree construction

Syntax tree for  $a - 4 + c$



# Application of SDD – Syntax tree construction

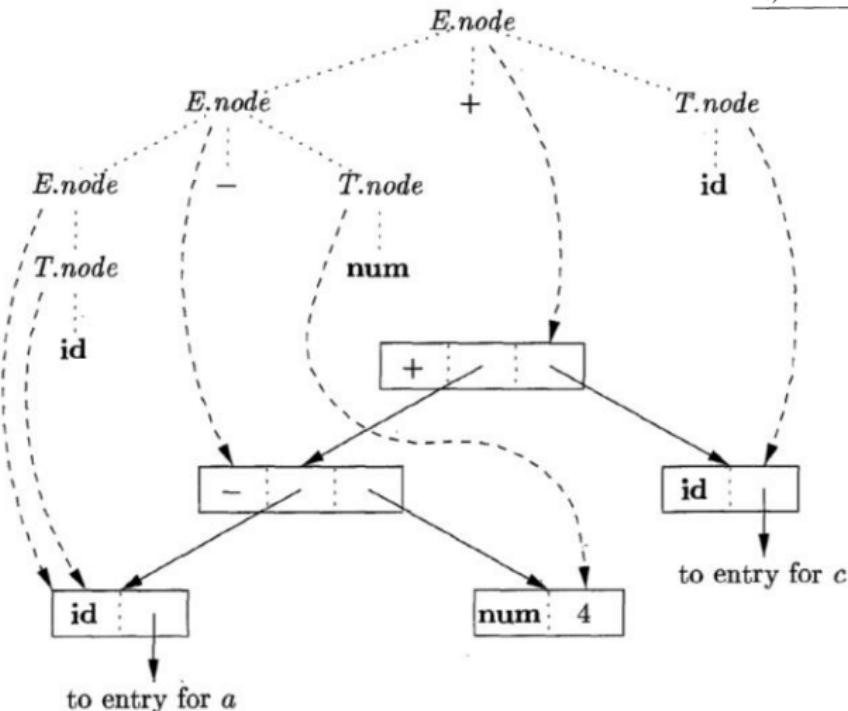
- Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
- A syntax-tree node representing an expression  $E_1 + E_2$  has label  $+$  and two children representing the subexpressions  $E_1$  and  $E_2$

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

# Application of SDD – Syntax tree construction

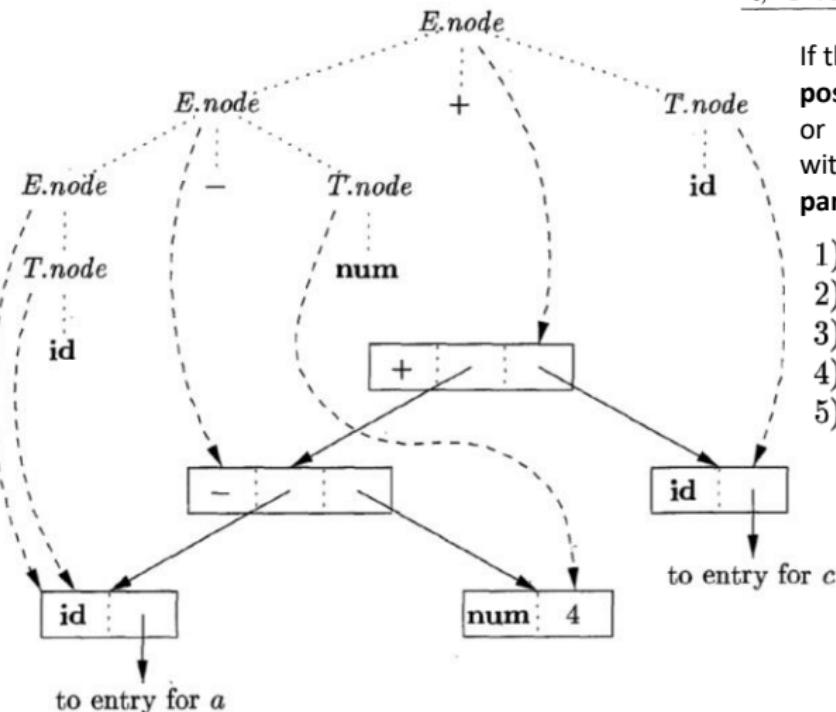
Syntax tree for  $a - 4 + c$

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



# Application of SDD – Syntax tree construction

Syntax tree for  $a - 4 + c$

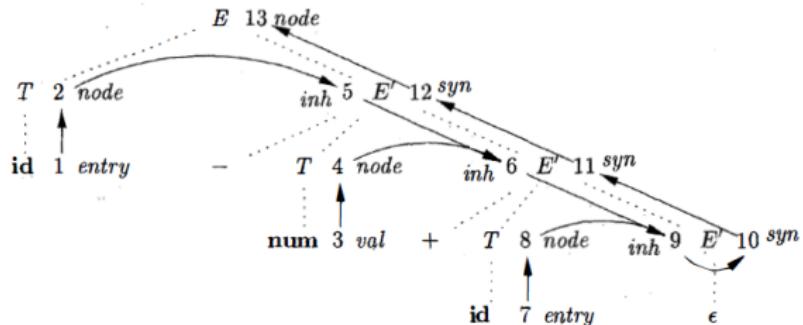


PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+' , E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow ( E )$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

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

- 1)  $p_1 = \text{new Leaf}(\text{id}, \text{entry-}a);$
- 2)  $p_2 = \text{new Leaf}(\text{num}, 4);$
- 3)  $p_3 = \text{new Node}('-', p_1, p_2);$
- 4)  $p_4 = \text{new Leaf}(\text{id}, \text{entry-}c);$
- 5)  $p_5 = \text{new Node}('+', p_3, p_4);$

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



Dependency graph for  $a - 4 + c$ ,

# Syntax-Directed Translation Schemes

- **Syntax-directed translation schemes** are a **complementary** notation to syntax directed definitions.
- All of the applications of syntax-directed definitions can be implemented using syntax-directed translation schemes.
- 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.

# Syntax-Directed Translation Schemes

$L \rightarrow E \ n$	{ $\text{print}(E.\text{val});$ }
$E \rightarrow E_1 + T$	{ $E.\text{val} = E_1.\text{val} + T.\text{val};$ }
$E \rightarrow T$	{ $E.\text{val} = T.\text{val};$ }
$T \rightarrow T_1 * F$	{ $T.\text{val} = T_1.\text{val} \times F.\text{val};$ }
$T \rightarrow F$	{ $T.\text{val} = F.\text{val};$ }
$F \rightarrow ( E )$	{ $F.\text{val} = E.\text{val};$ }
$F \rightarrow \text{digit}$	{ $F.\text{val} = \text{digit}.\text{lexval};$ }

- 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**
  - **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**.

# Syntax-Directed Translation Schemes

$L \rightarrow E \text{ n}$	{ $\text{print}(E.\text{val});$ }
$E \rightarrow E_1 + T$	{ $E.\text{val} = E_1.\text{val} + T.\text{val};$ }
$E \rightarrow T$	{ $E.\text{val} = T.\text{val};$ }
$T \rightarrow T_1 * F$	{ $T.\text{val} = T_1.\text{val} \times F.\text{val};$ }
$T \rightarrow F$	{ $T.\text{val} = F.\text{val};$ }
$F \rightarrow ( E )$	{ $F.\text{val} = E.\text{val};$ }
$F \rightarrow \text{digit}$	{ $F.\text{val} = \text{digit}.\text{lexval};$ }

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker  $M$  has only one production,  $M \rightarrow \epsilon$ . If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing.

# Parser Implementation of Postfix SDT's

- Postfix SDT's can be implemented during **LR parsing** by executing the **actions when reductions occur**.
- The **attribute(s)** of each grammar symbol can be put **on the stack** in a place where they can be found during the reduction.
- The parser stack contains **records** with a field for a **grammar symbol (or parser state)** and, below it, a **field for an attribute**

X	Y	Z
X.x	Y.y	Z.z

↑  
top

State/grammar symbol  
Synthesized attribute(s)

The three grammar symbols X Y Z  
are on top of the stack.

Perhaps they are about to be  
**reduced** according to a production  
like A->X Y Z

# Parser Implementation of Postfix SDT's

X	Y	Z
X.x	Y.y	Z.z

State/grammar symbol

Synthesized attribute(s)

↑  
top

A->X Y Z

- If the attributes are **all synthesized**, and the **actions occur at the ends** of the productions
  - then we can **compute the attributes** for the **head** when we **reduce the body** to the head.
- If we **reduce by a production** such as A -> X Y Z, then
  - we have all the **attributes** of X, Y, and Z available, at **known positions** on the stack.
- **After the action, A and its attributes are pushed** at the top of the stack, in the position of the **record for X**

## Actions of the desk-calculator SDT so that they manipulate the parser stack explicitly

$L \rightarrow E \ n$	{ print( $E.val$ ); }
$E \rightarrow E_1 + T$	{ $E.val = E_1.val + T.val;$ }
$E \rightarrow T$	{ $E.val = T.val;$ }
$T \rightarrow T_1 * F$	{ $T.val = T_1.val \times F.val;$ }
$T \rightarrow F$	{ $T.val = F.val;$ }
$F \rightarrow ( E )$	{ $F.val = E.val;$ }
$F \rightarrow \text{digit}$	{ $F.val = \text{digit}.lexval;$ }

PRODUCTION      ACTIONS

$L \rightarrow E \ n$       { print( $stack[top - 1].val$ );  
 $top = top - 1;$  }

$E \rightarrow E_1 + T$       {  $stack[top - 2].val = stack[top - 2].val + stack[top].val;$   
 $top = top - 2;$  }

$E \rightarrow T$

$T \rightarrow T_1 * F$       {  $stack[top - 2].val = stack[top - 2].val \times stack[top].val;$   
 $top = top - 2;$  }

$T \rightarrow F$

$F \rightarrow ( E )$       {  $stack[top - 2].val = stack[top - 1].val;$   
 $top = top - 2;$  }

$F \rightarrow \text{digit}$

# SDT's With Actions 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  $B \rightarrow X \{a\} Y$ , 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 for infix-to-prefix translation during parsing

$3 * 5 + 4$

$+ * 3 5 4.$

- 1)  $L \rightarrow E \text{ n}$
- 2)  $E \rightarrow \{ \text{print}('+'); \} E_1 + T$
- 3)  $E \rightarrow T$
- 4)  $T \rightarrow \{ \text{print}('*'); \} T_1 * F$
- 5)  $T \rightarrow F$
- 6)  $F \rightarrow ( E )$
- 7)  $F \rightarrow \text{digit } \{ \text{print}(\text{digit}.lexval); \}$

- It is impossible to implement this SDT during either topdown or bottom-up parsing,
  - because the parser would have to perform semantic actions, like printing instances of  $*$  or  $+$ ,
  - long before it knows whether these symbols will appear in its input .

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node  $N$ , say one for production  $A \rightarrow \alpha$ . Add additional children to  $N$  for the actions in  $\alpha$ , so the children of  $N$  from left to right have exactly the symbols and actions of  $\alpha$ .
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

parse tree for expression  $3 * 5 + 4$

If we visit the nodes in preorder,  
we get the prefix form of the  
expression: **+ \* 3 5 4**

