

# Chapter-05

## Syntax-Directed Translation

*Compilers: Principles, Techniques, & Tools, Second Edition*

*Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman*

# Outline

- ❑ Syntax-Directed Definitions
  - ❑ Inherited and Synthesized Attributes
  - ❑ Evaluating an SDD at the Nodes of a Parse Tree
- ❑ Evaluation Orders for SDD's
  - ❑ Dependency Graphs
  - ❑ Ordering the Evaluation of Attributes
  - ❑ S-Attributed Definitions
  - ❑ L-Attributed Definitions
  - ❑ Semantic Rules with Controlled Side Effects
- ❑ Applications of Syntax-Directed Translation
- ❑ Syntax-Directed Translation Schemes

# Syntax-Directed Definitions

# 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

# Attributes for Nonterminals

- Two kinds of attributes for non-terminals:
  - 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
- ❑ 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
- 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) $L \rightarrow E \text{ n}$	$L.val = E.val$
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) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

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 production rules of an SDD are applied by first constructing a parse tree
  - ✓ Then using the semantic 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 the **bottom-up order**, such as that of a post order traversal of the parse tree

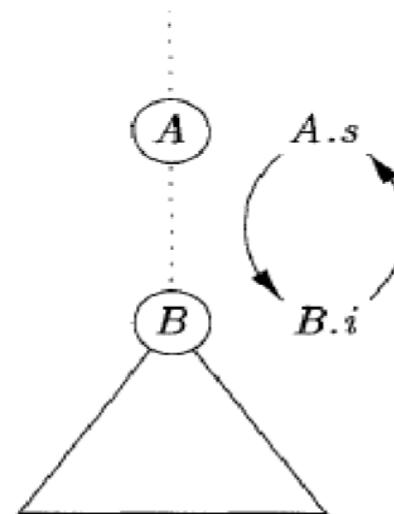
# Circular Dependency

- 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
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

# 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 some useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists.

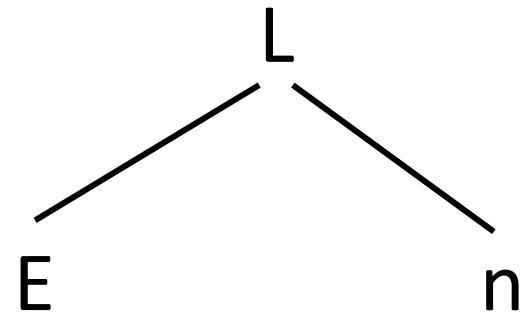
## Example 5.2 Annotated Parse Tree for $3 * 5 + 4$ n

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

Syntax-directed definition of a simple desk calculator

## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$



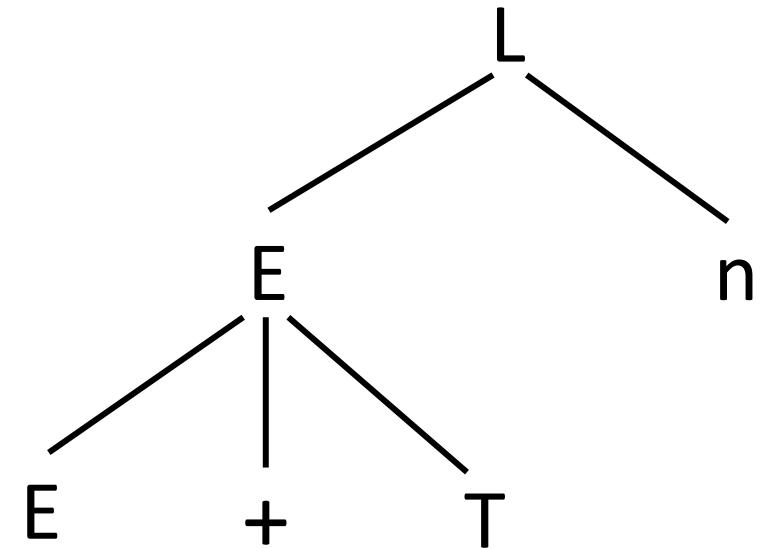
PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \text{ n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow ( E )$	$F.val = E.val$
7)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

Syntax-directed definition of a simple desk calculator

## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

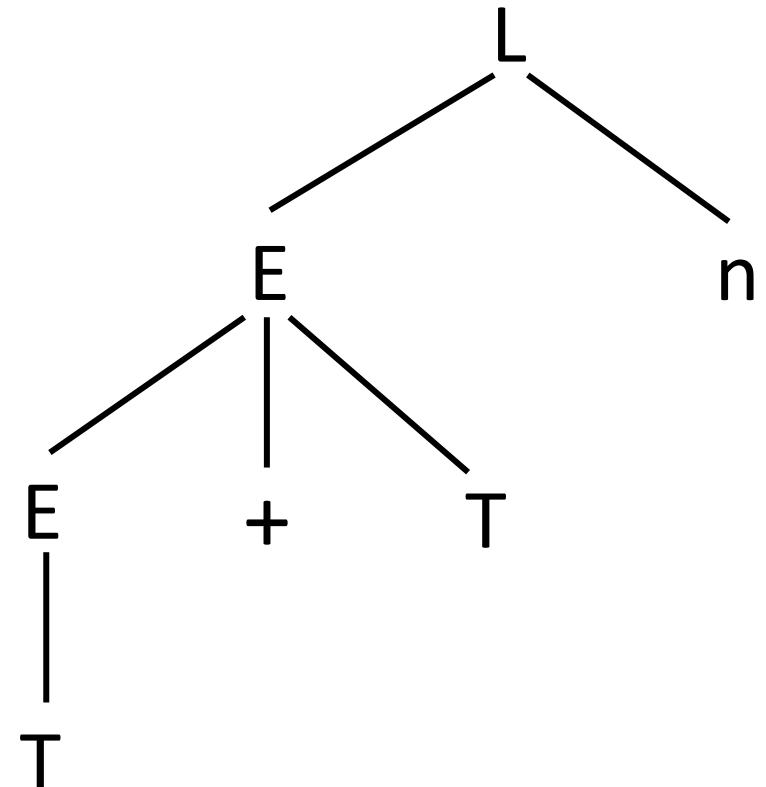


Syntax-directed definition of a simple desk calculator

## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

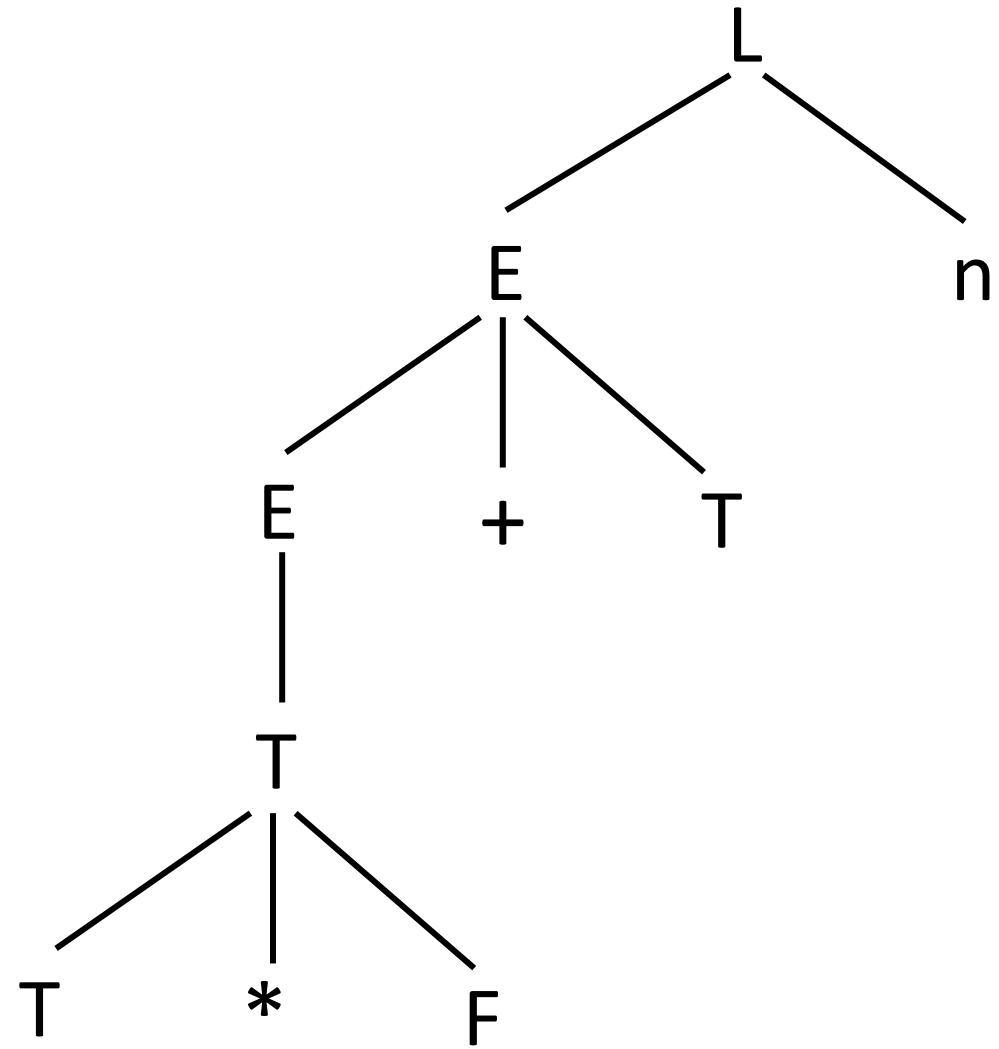


Syntax-directed definition of a simple desk calculator

## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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



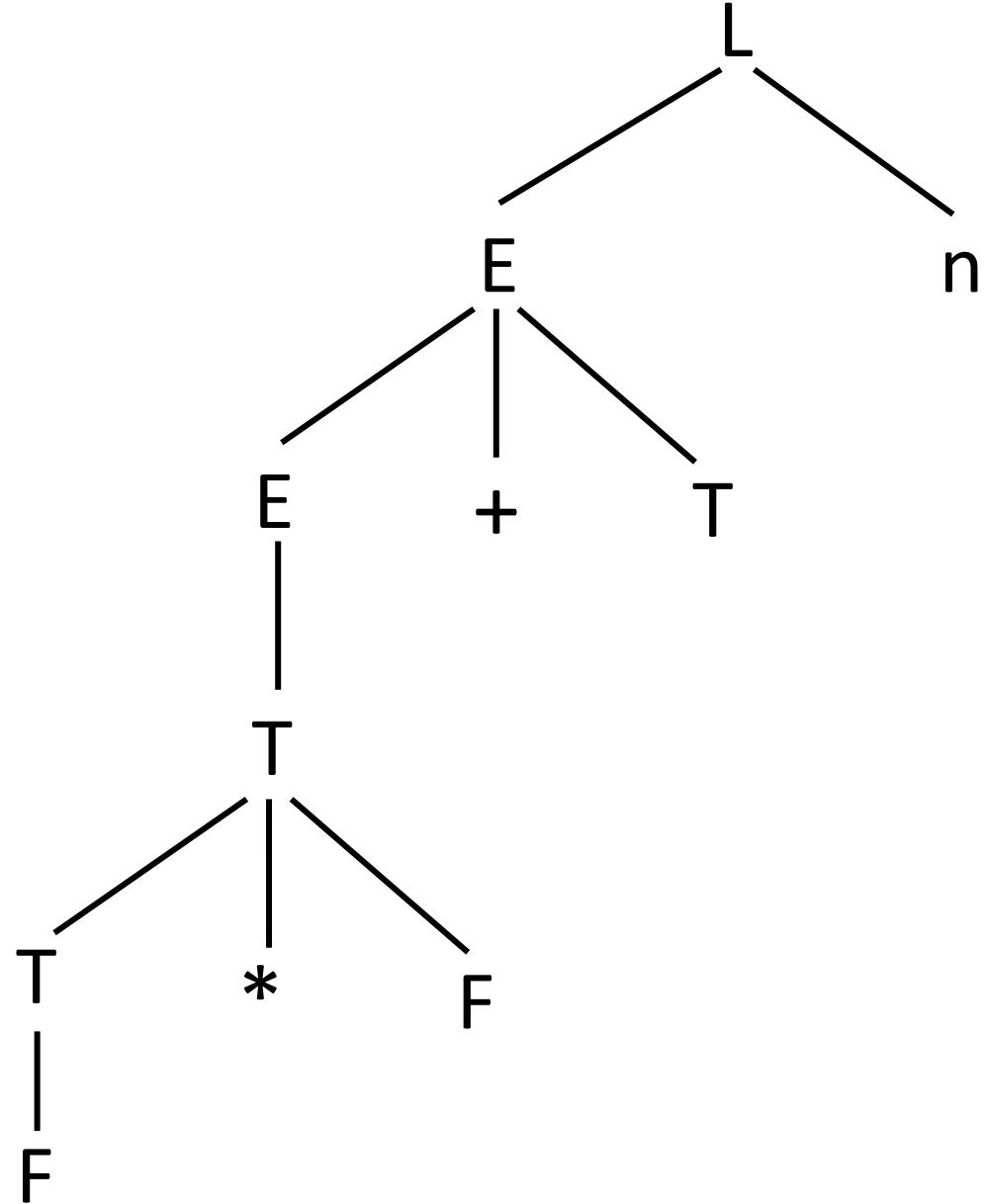
Syntax-directed definition of a simple desk calculator

## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

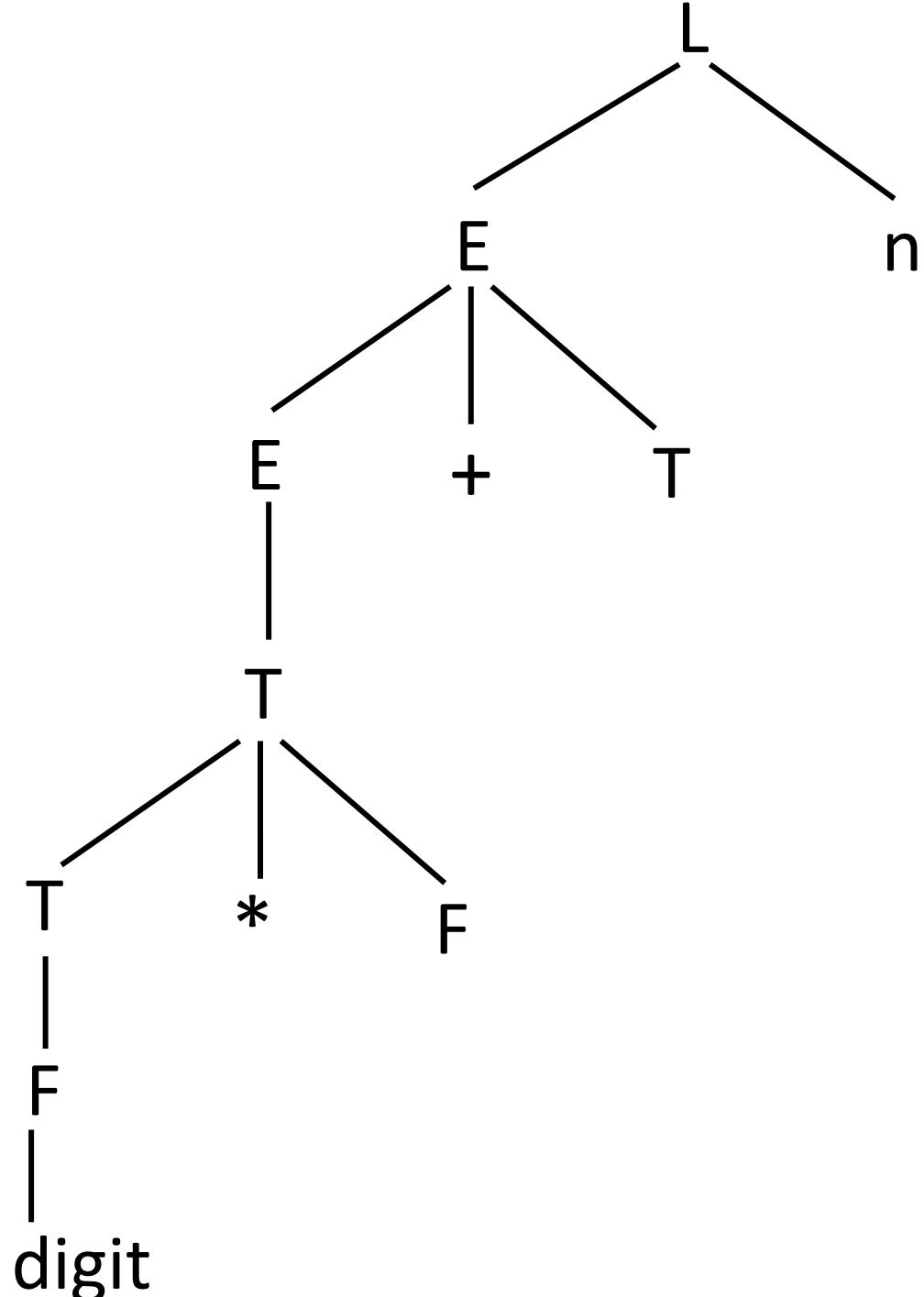


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

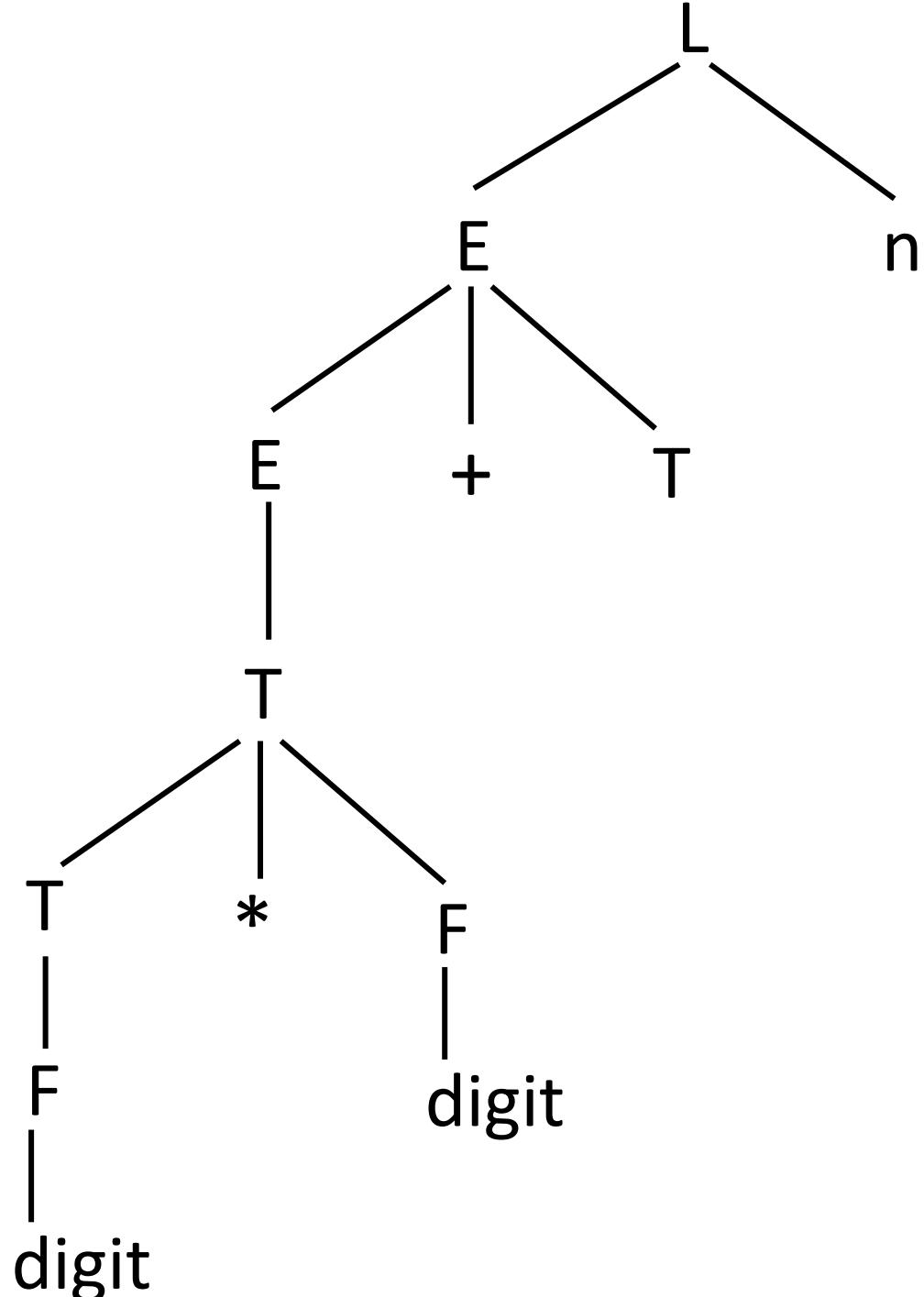


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

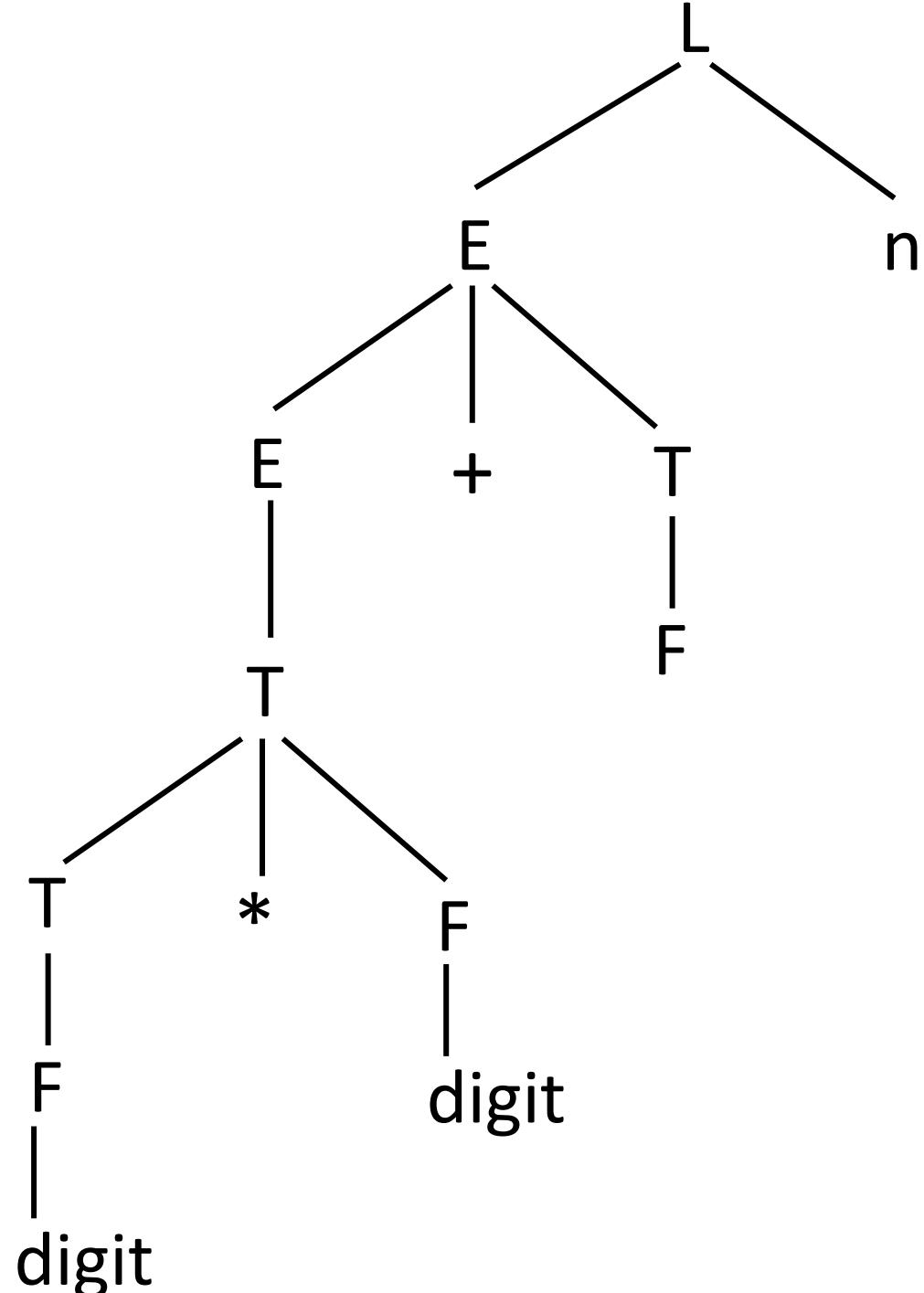


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

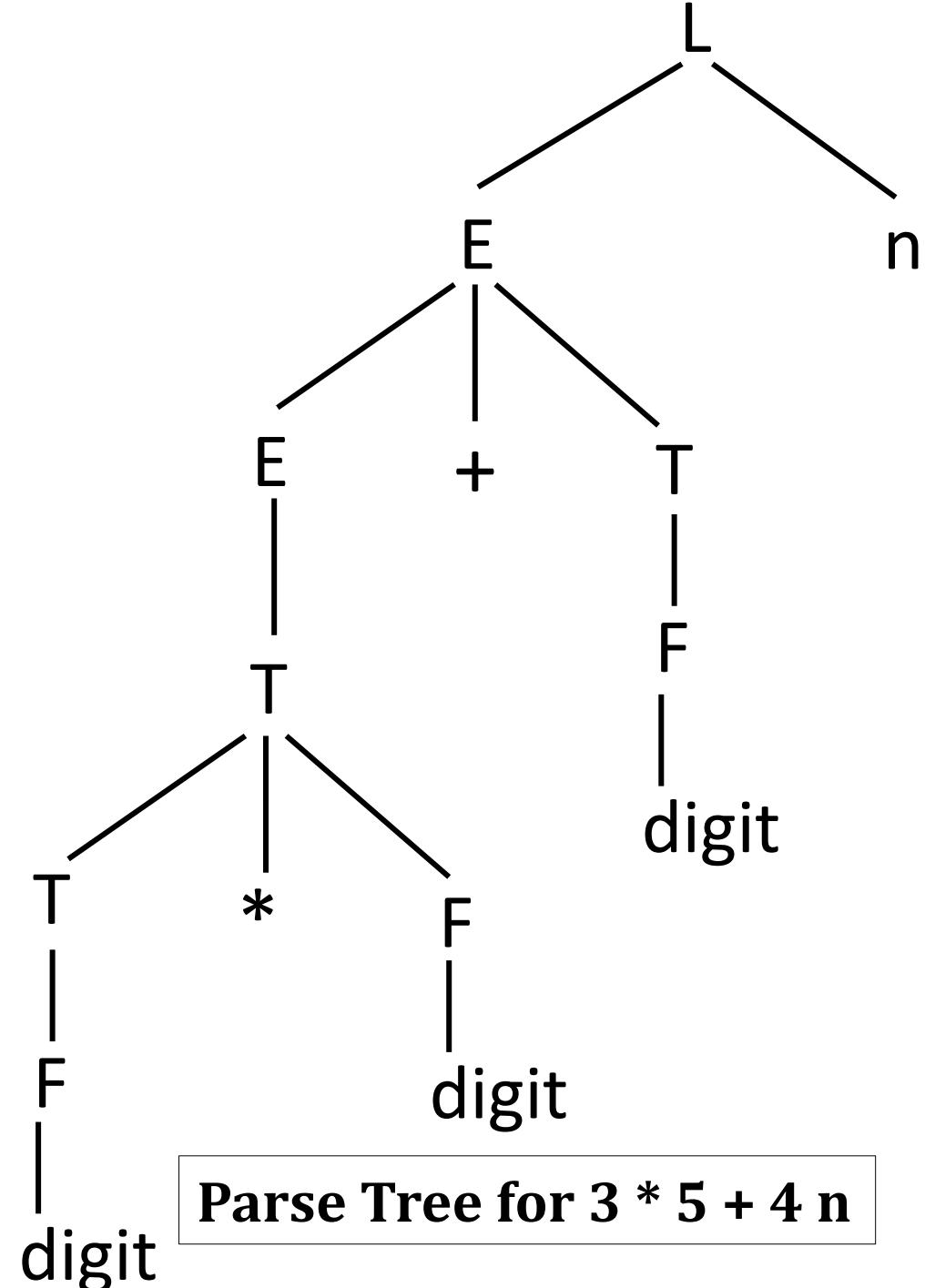


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

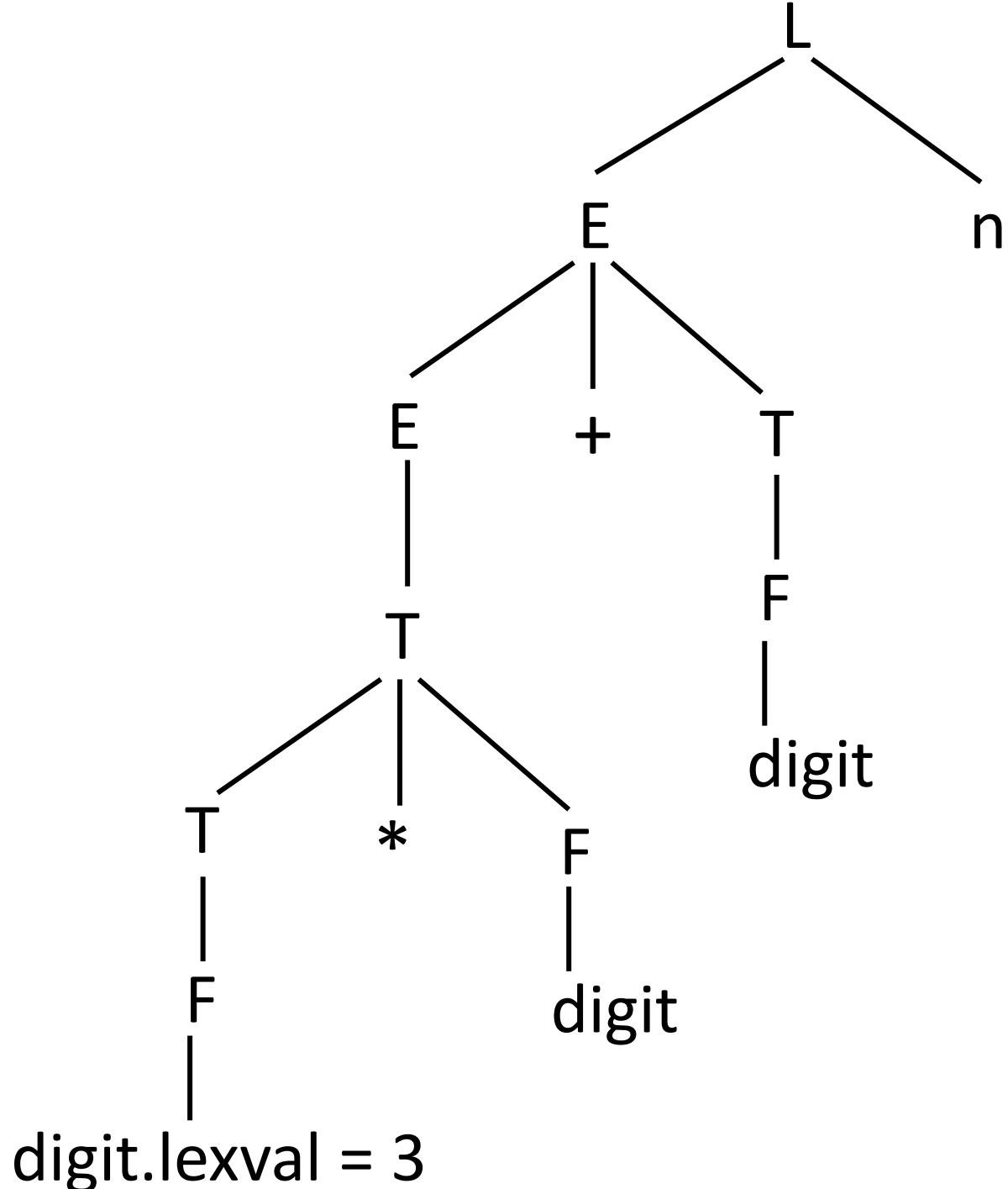


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

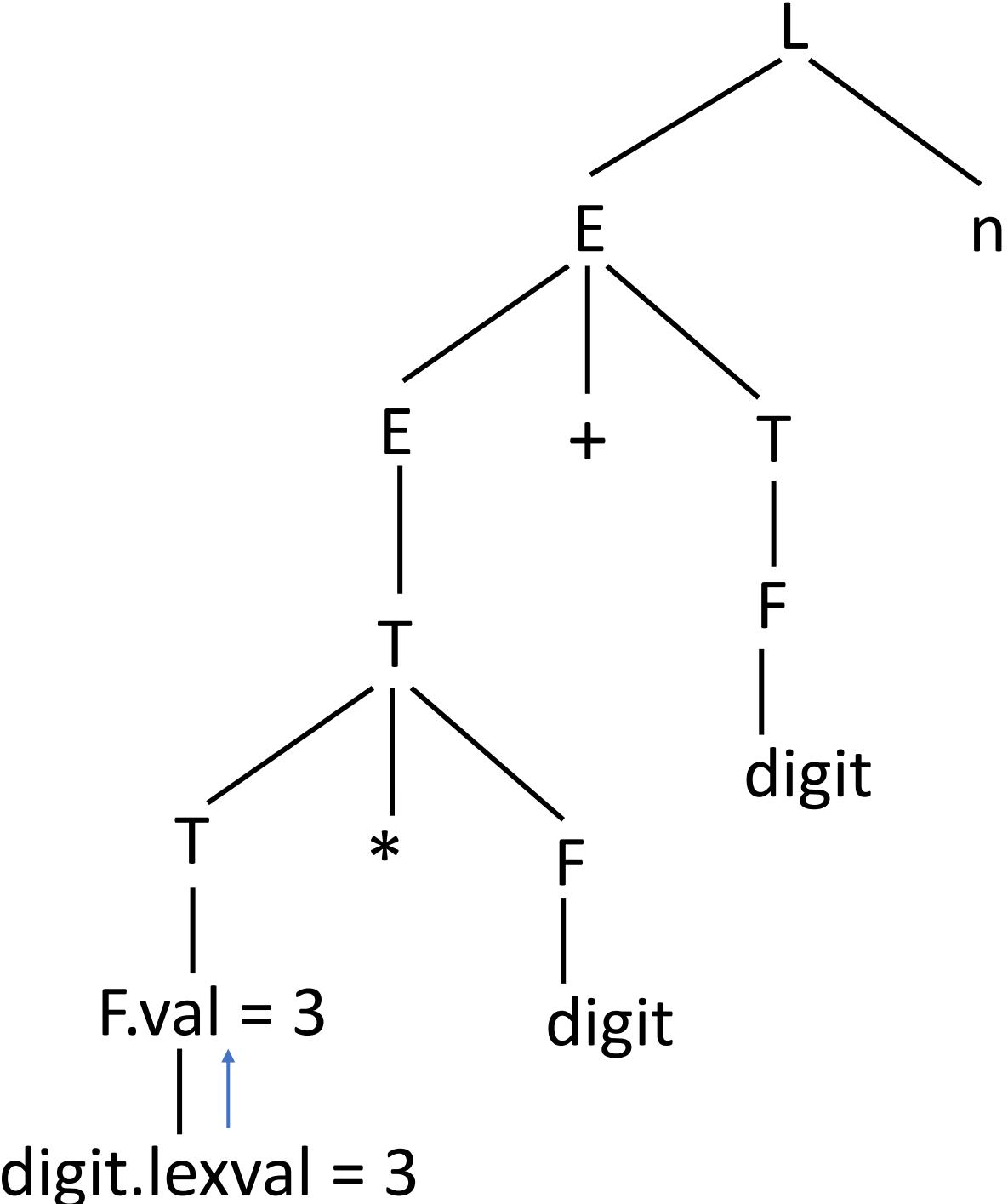


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

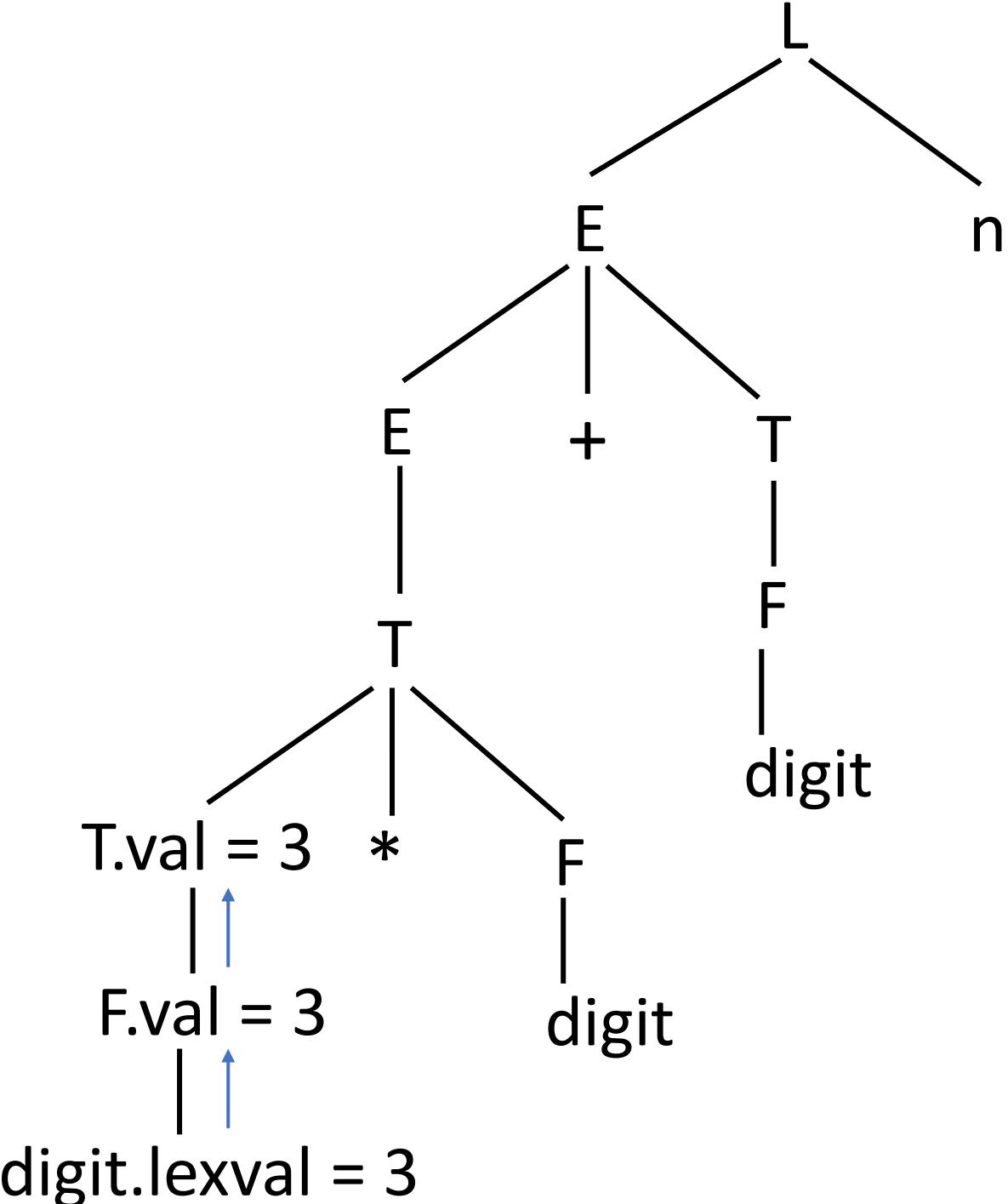


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

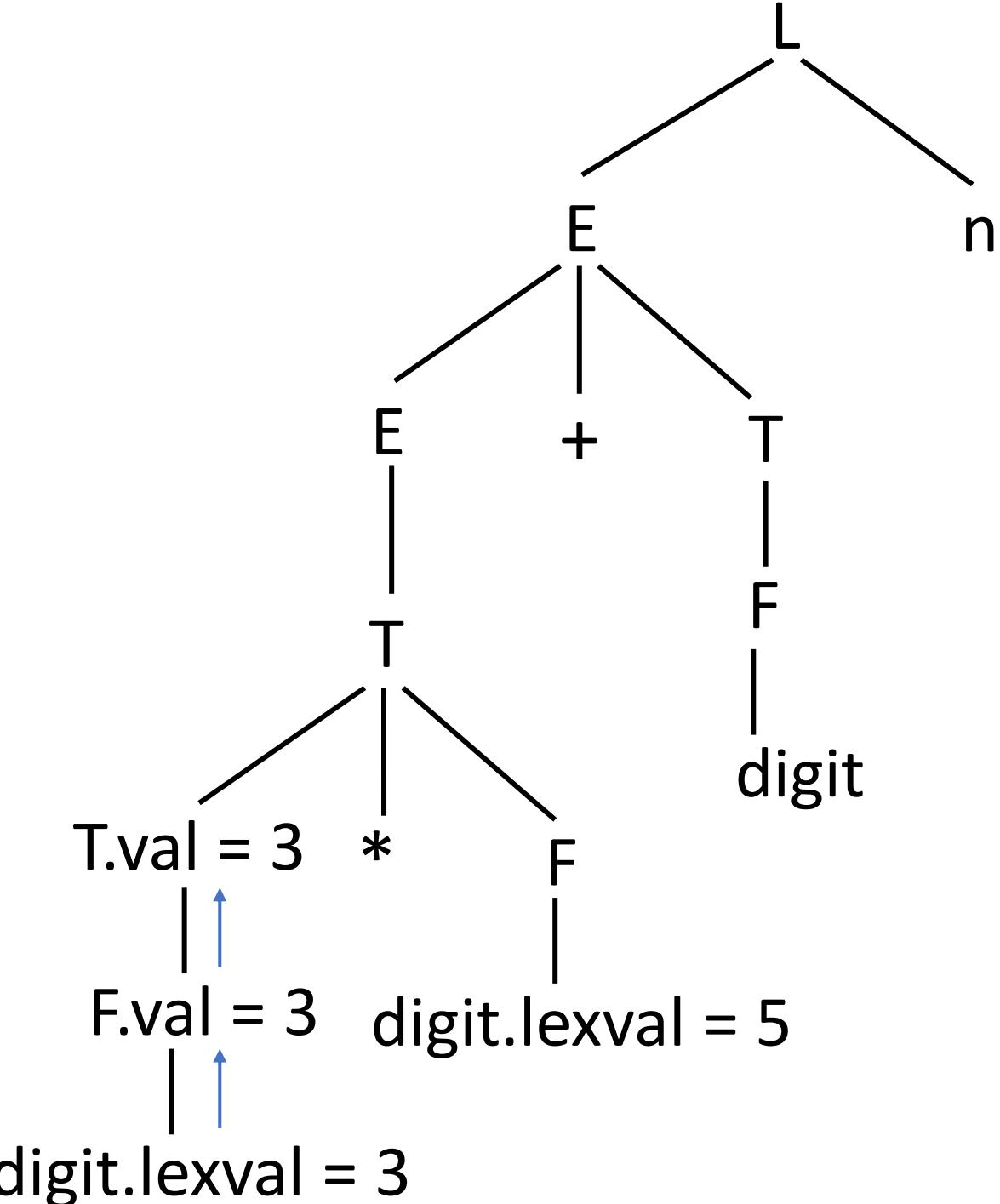


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

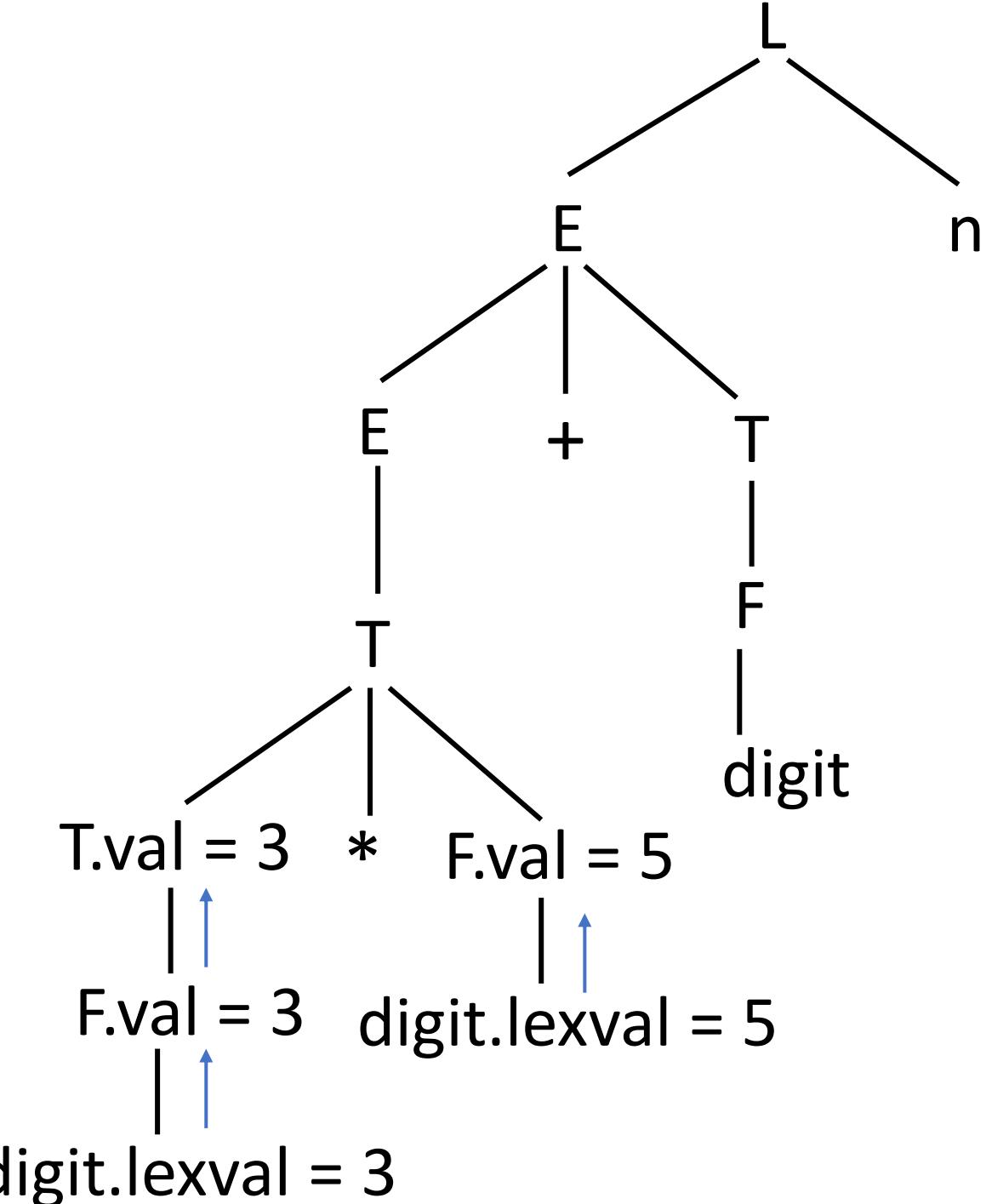


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

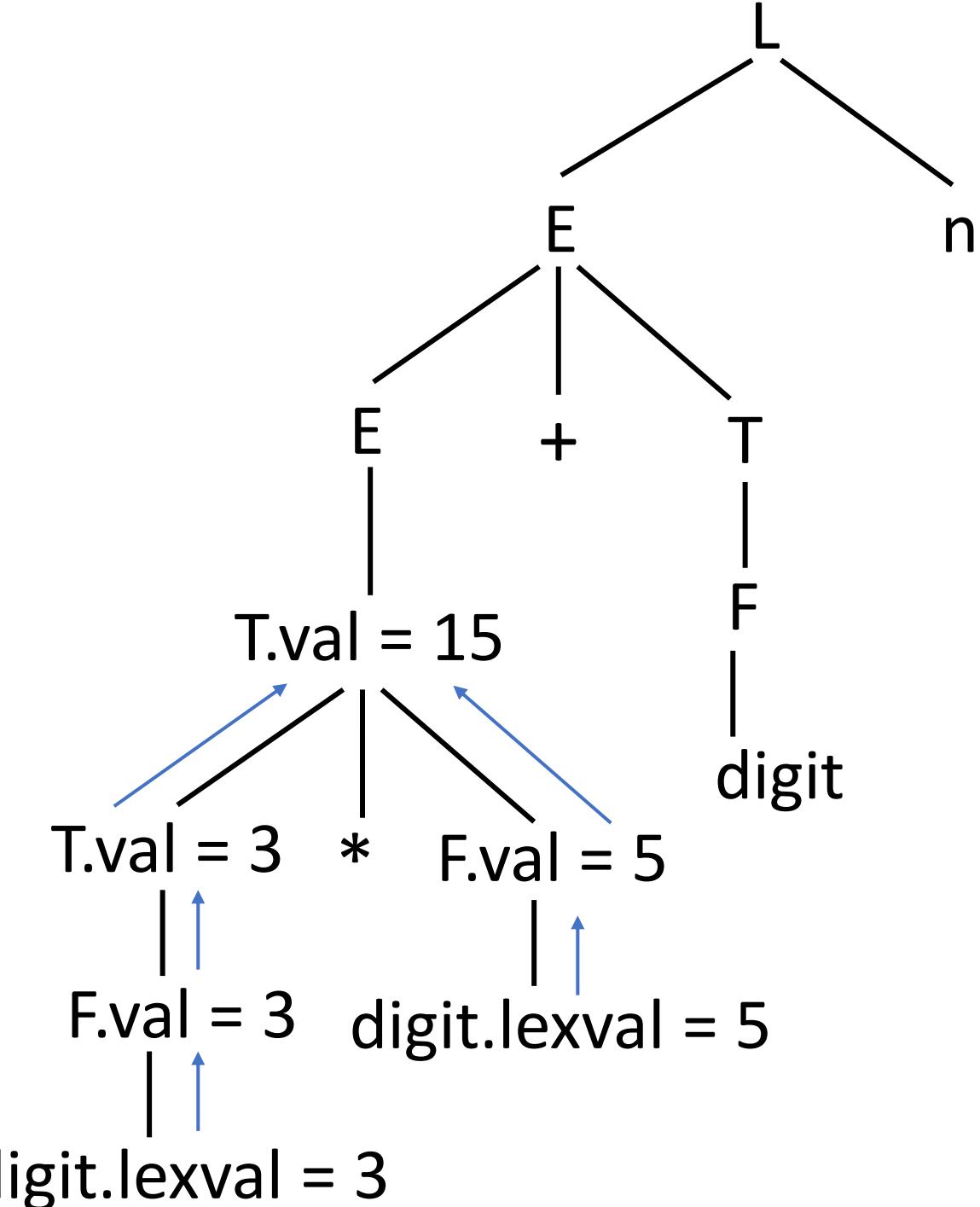


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

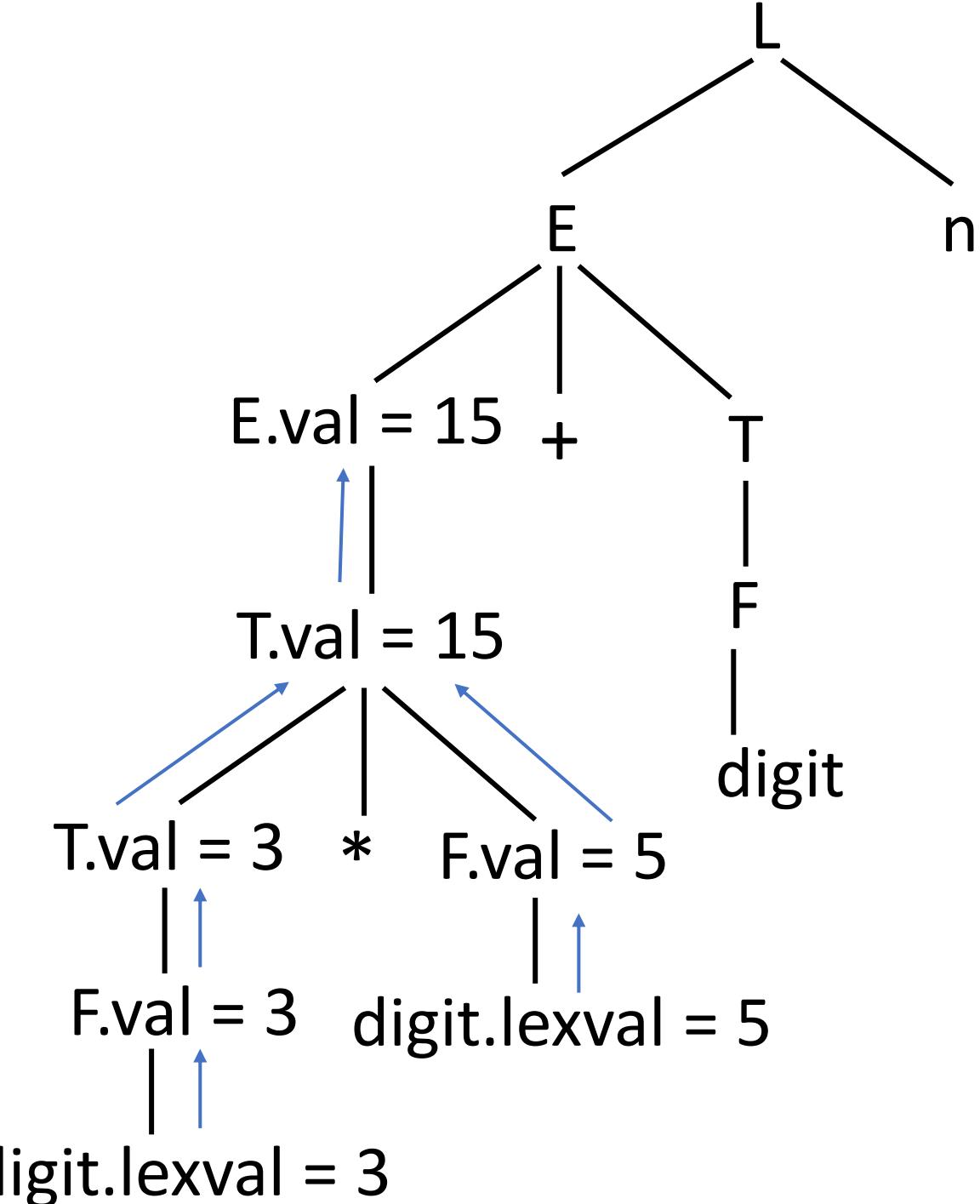


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator

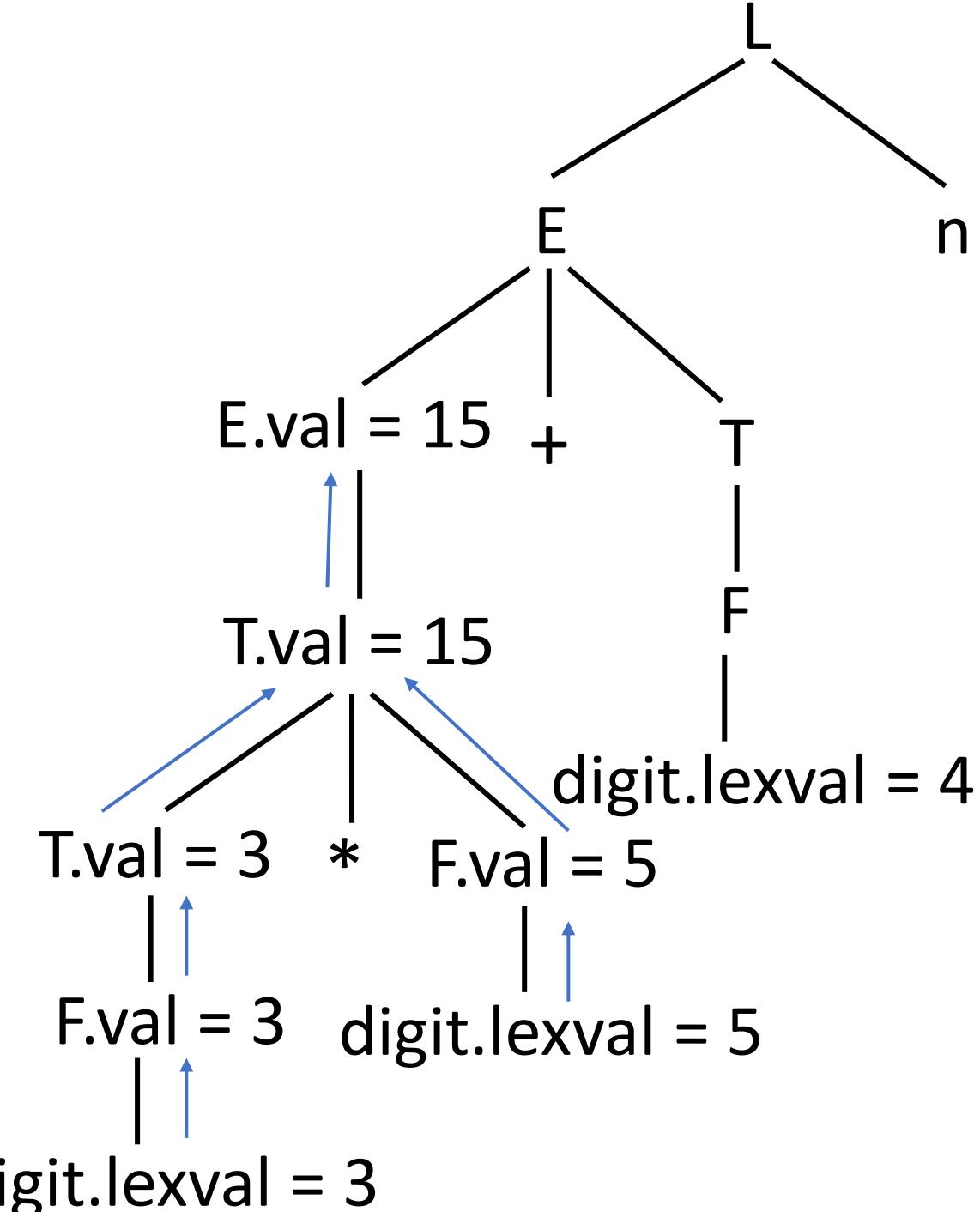


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

Syntax-directed definition of a simple desk calculator

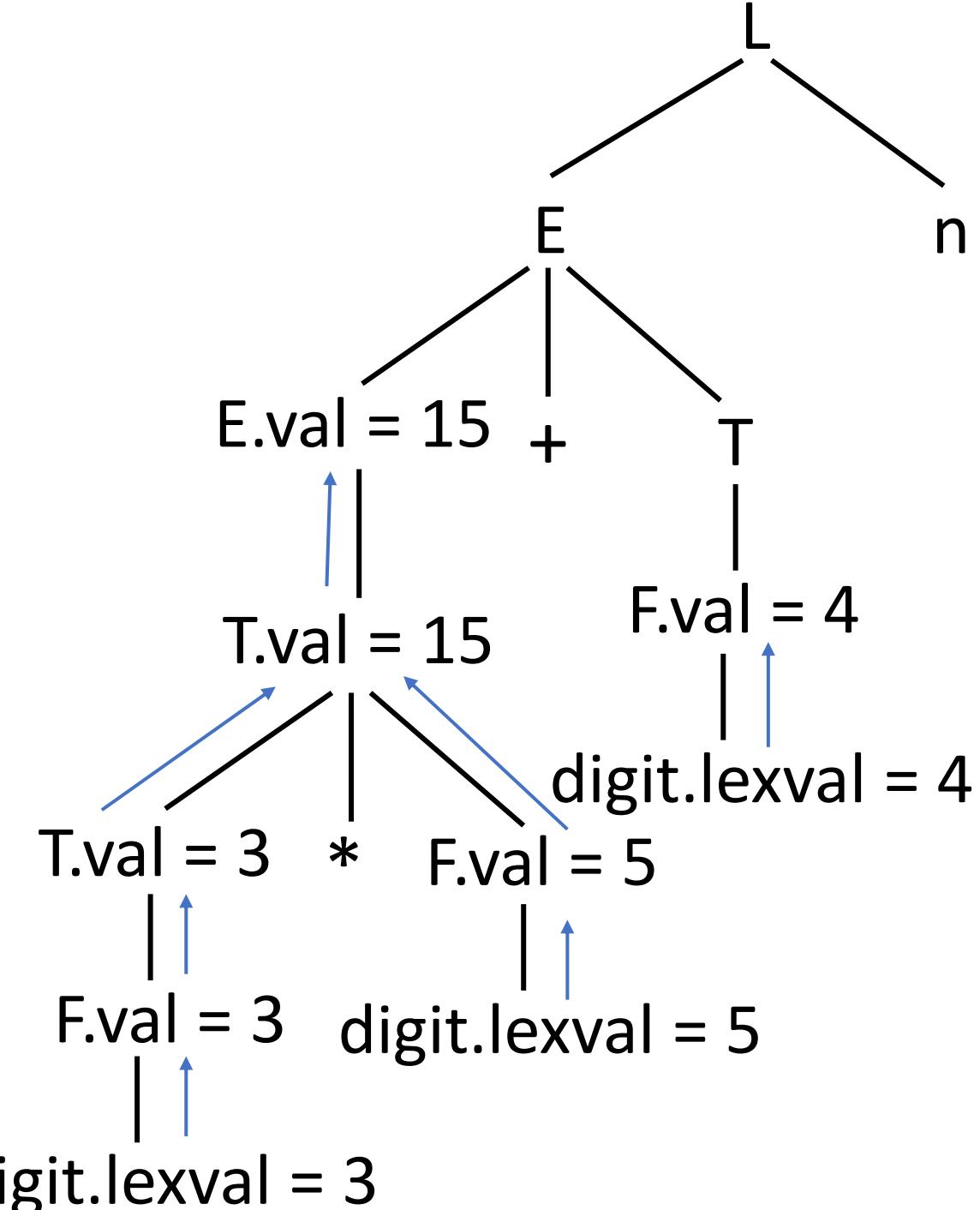


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

Syntax-directed definition of a simple desk calculator

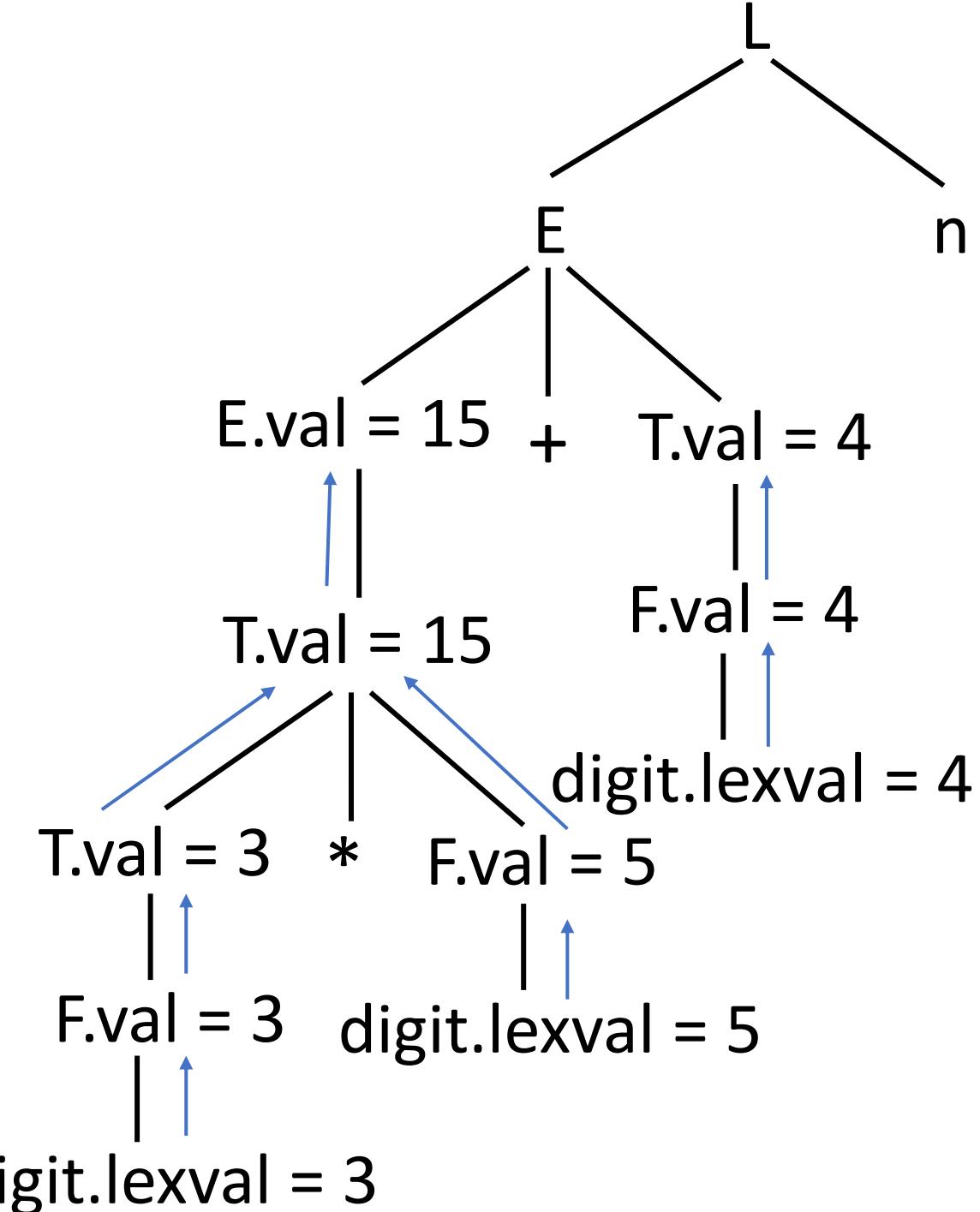


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

Syntax-directed definition of a simple desk calculator

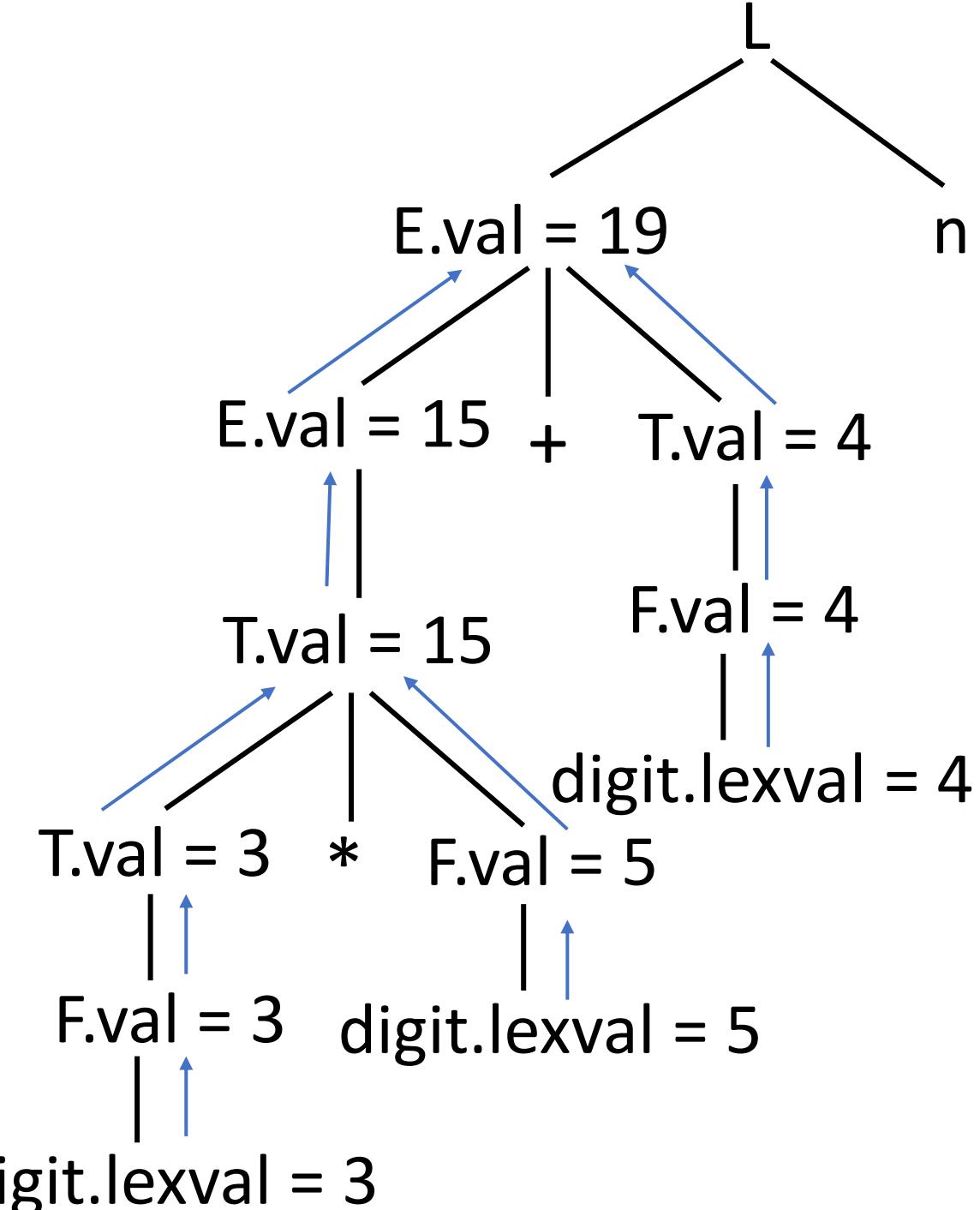


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 \text{ n}$

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

Syntax-directed definition of a simple desk calculator

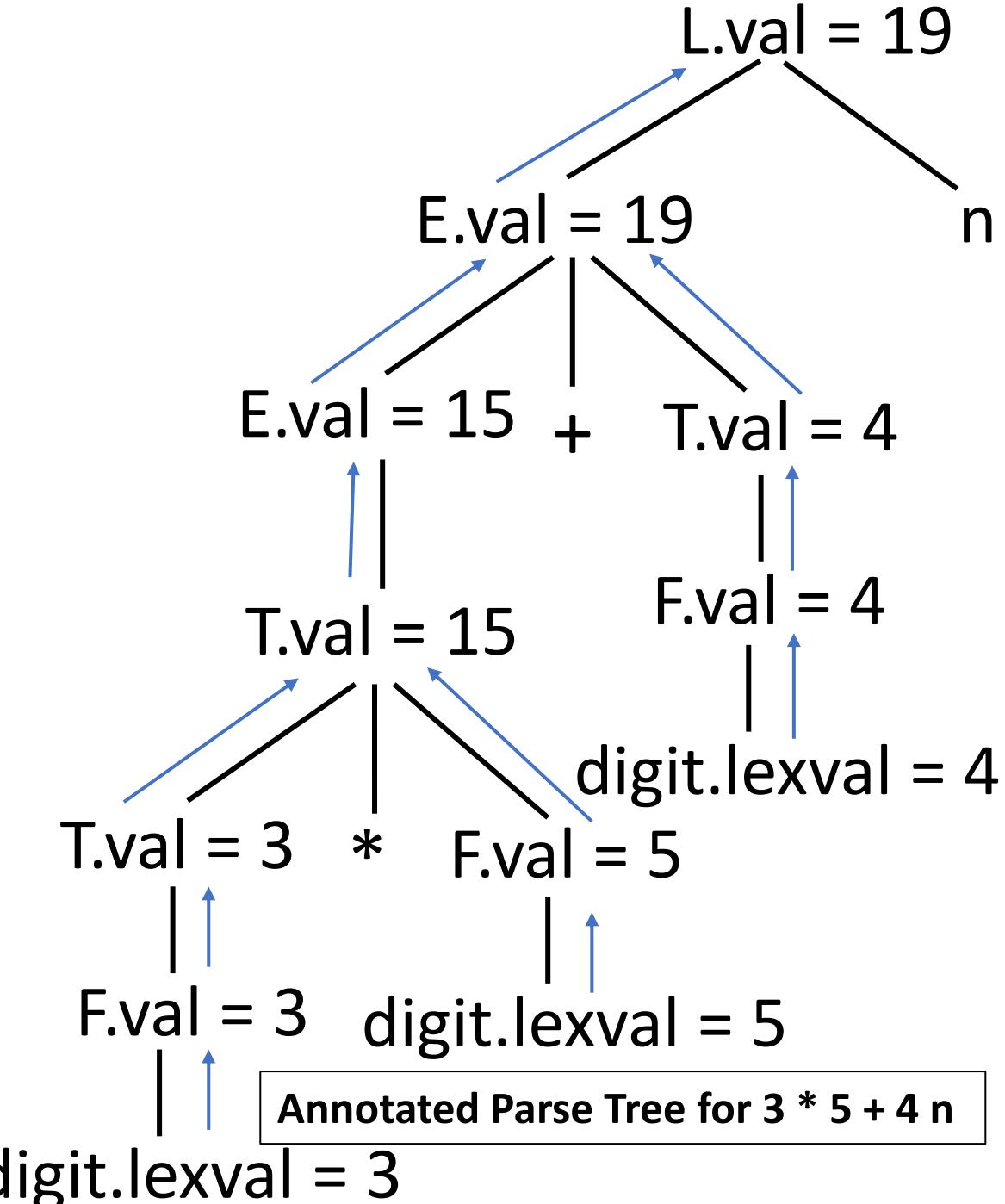


## Example 5.2 (Cont...)

Annotated Parse Tree for  $3 * 5 + 4 n$

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

Syntax-directed definition of a simple desk calculator



# Example 5.3

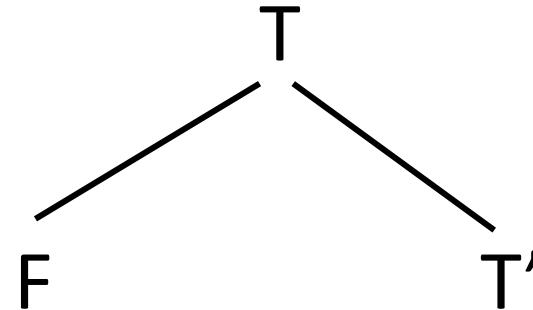
Annotated Parse Tree for  $3 * 5$

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

SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$



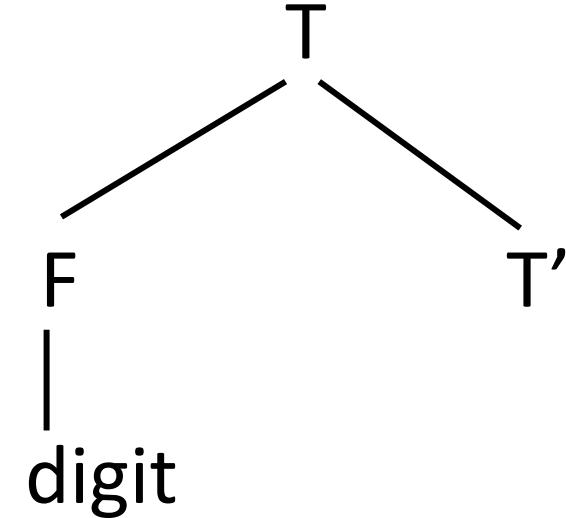
PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

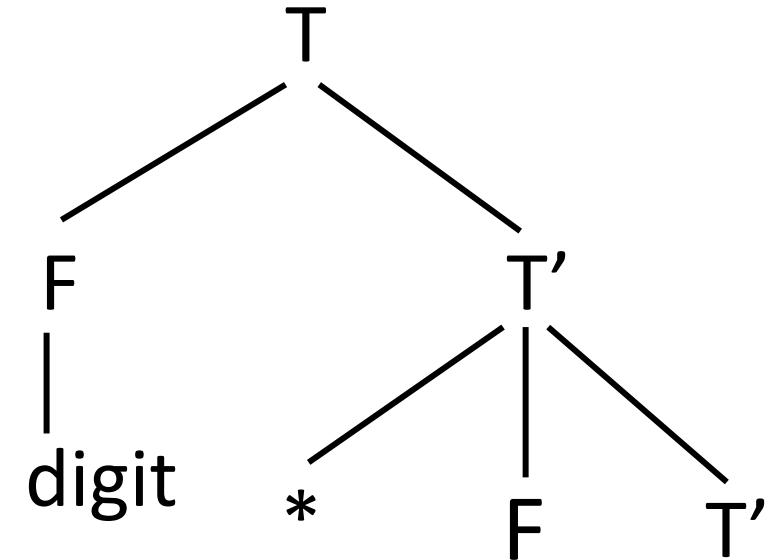


SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

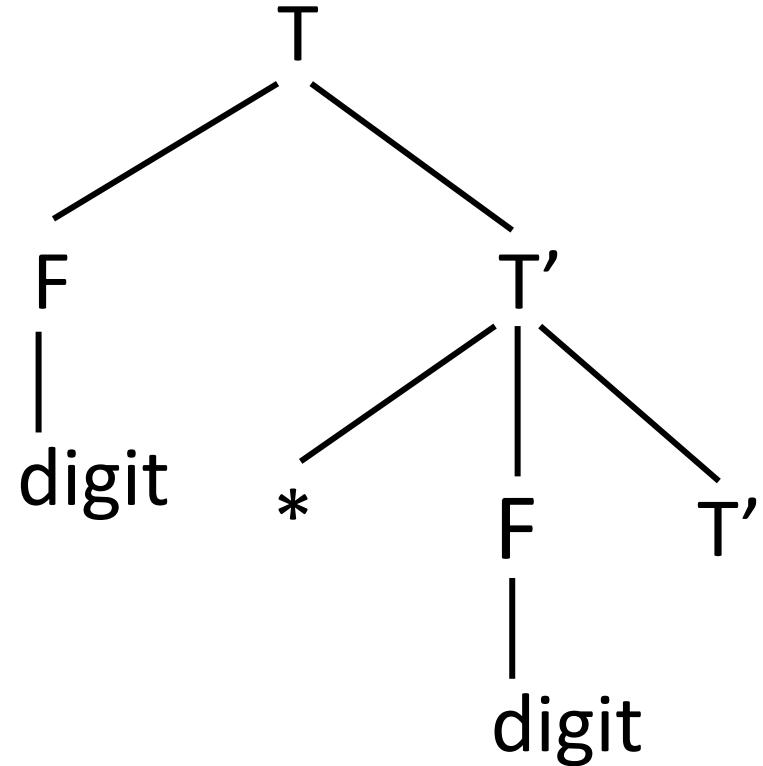


SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

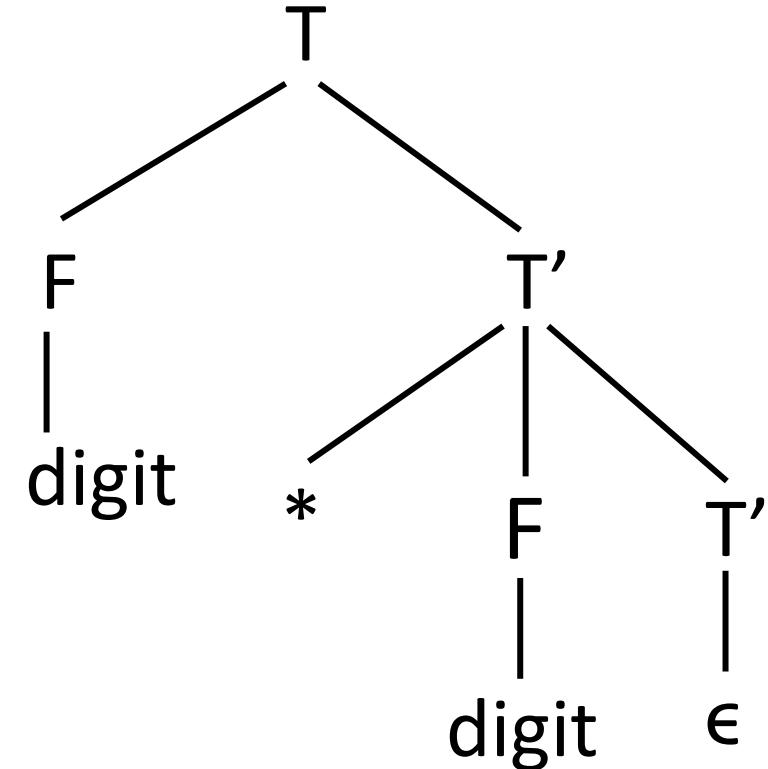


SDD based on a grammar suitable for top-down parsing

# Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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



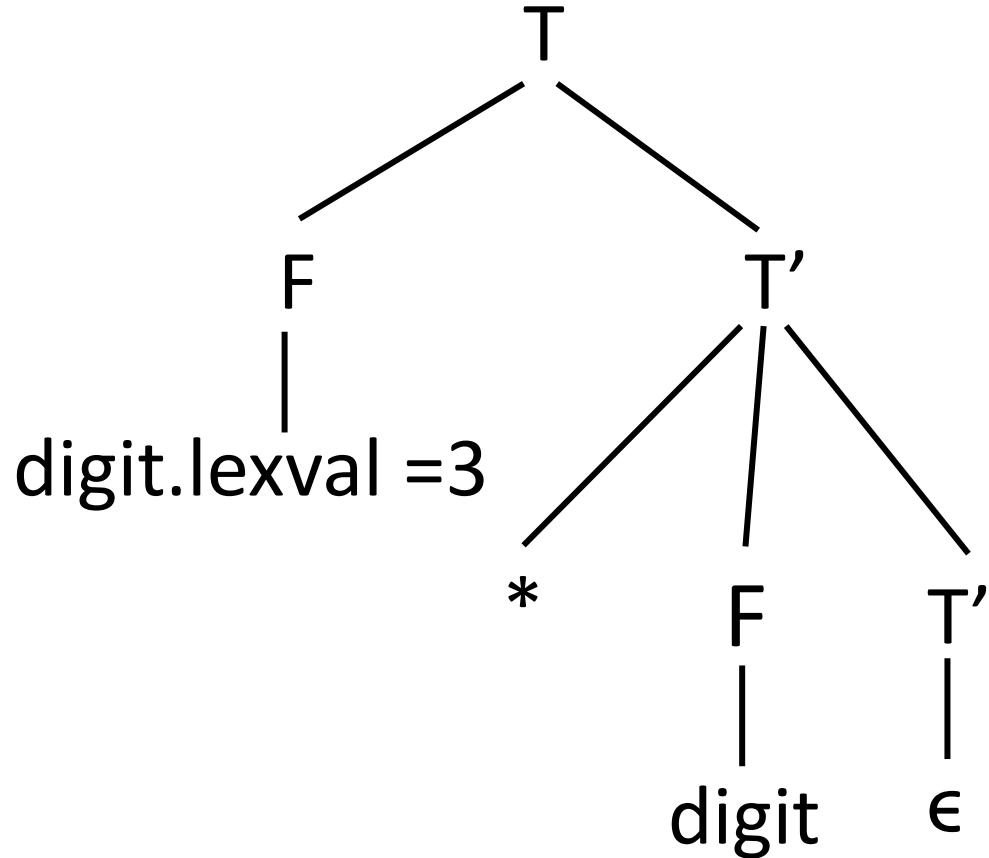
Parse Tree for  $3 * 5$

SDD based on a grammar suitable for top-down parsing

# Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

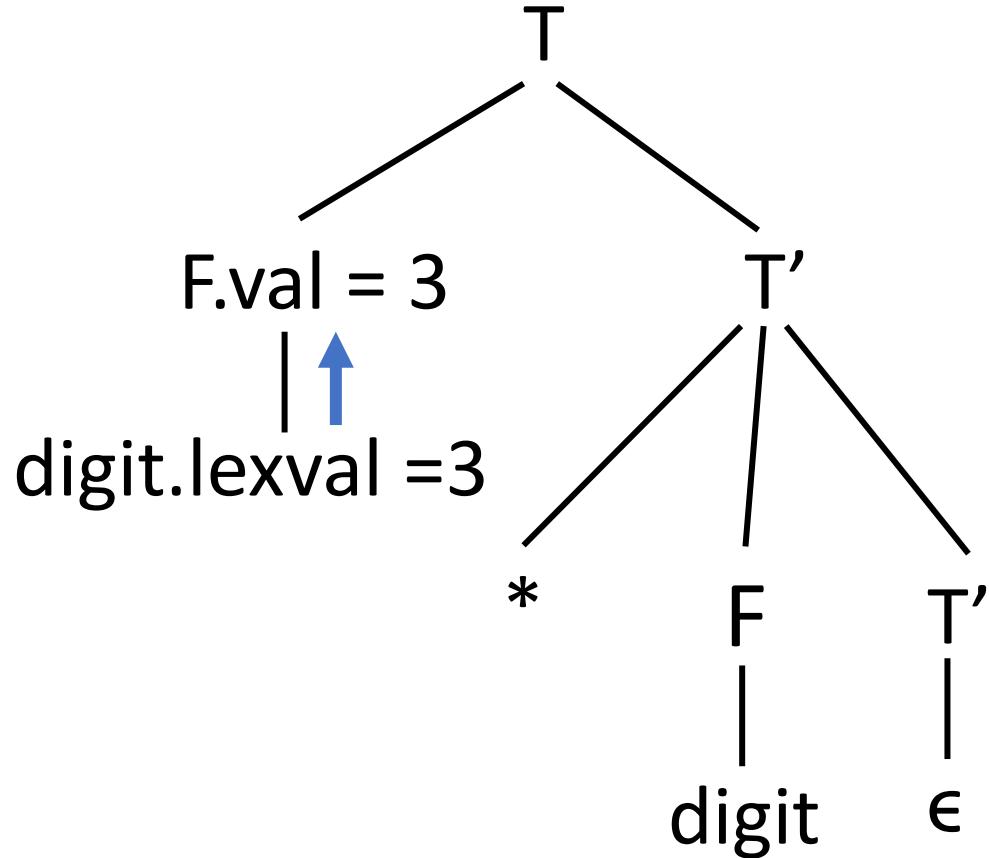


SDD based on a grammar suitable for top-down parsing

# Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

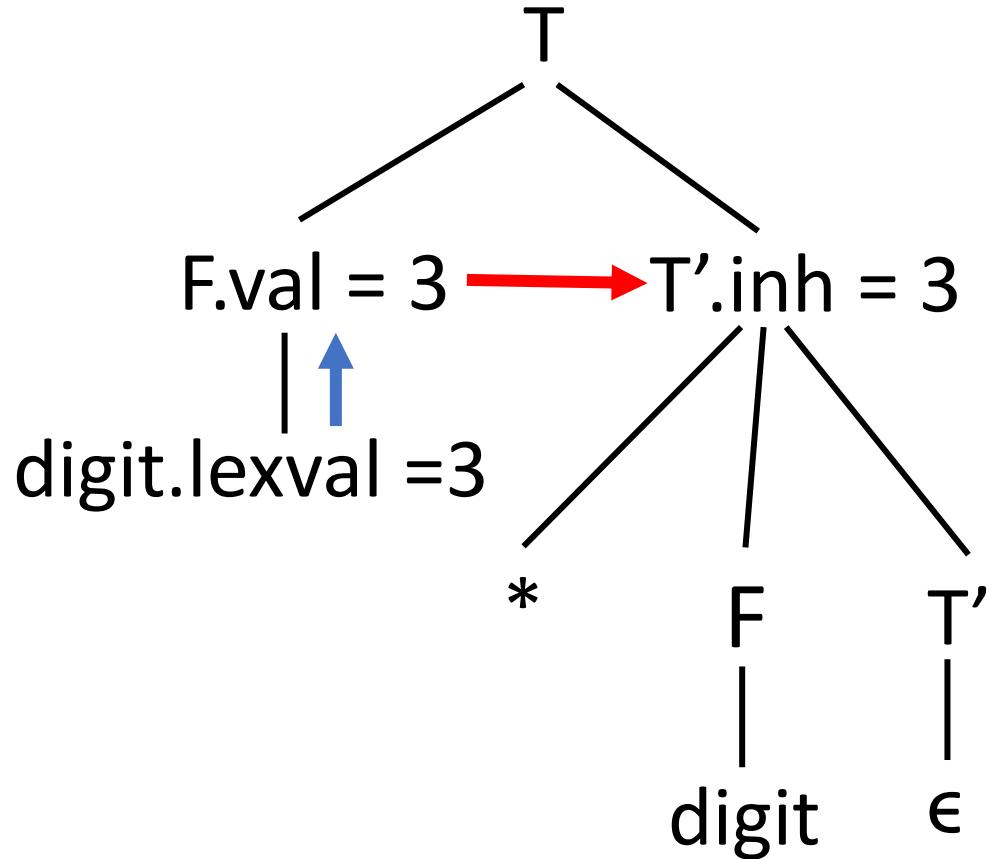


SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

## Annotated Parse Tree for $3 * 5$

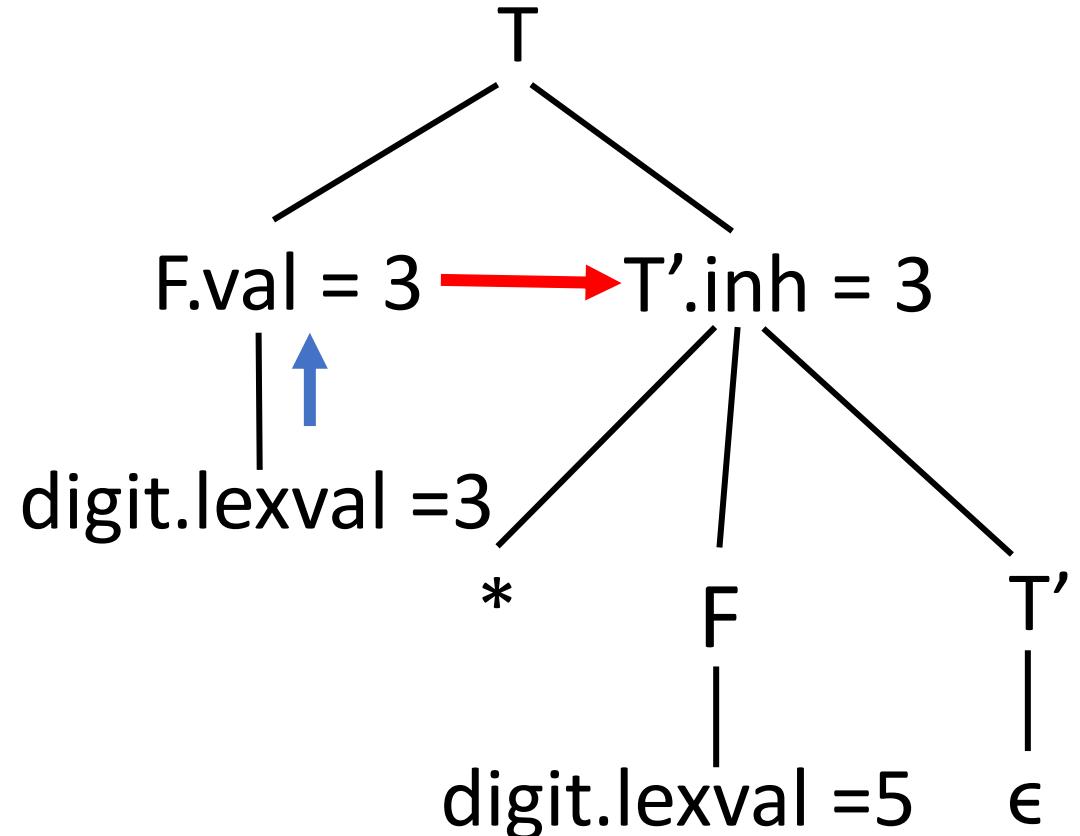
PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$



## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

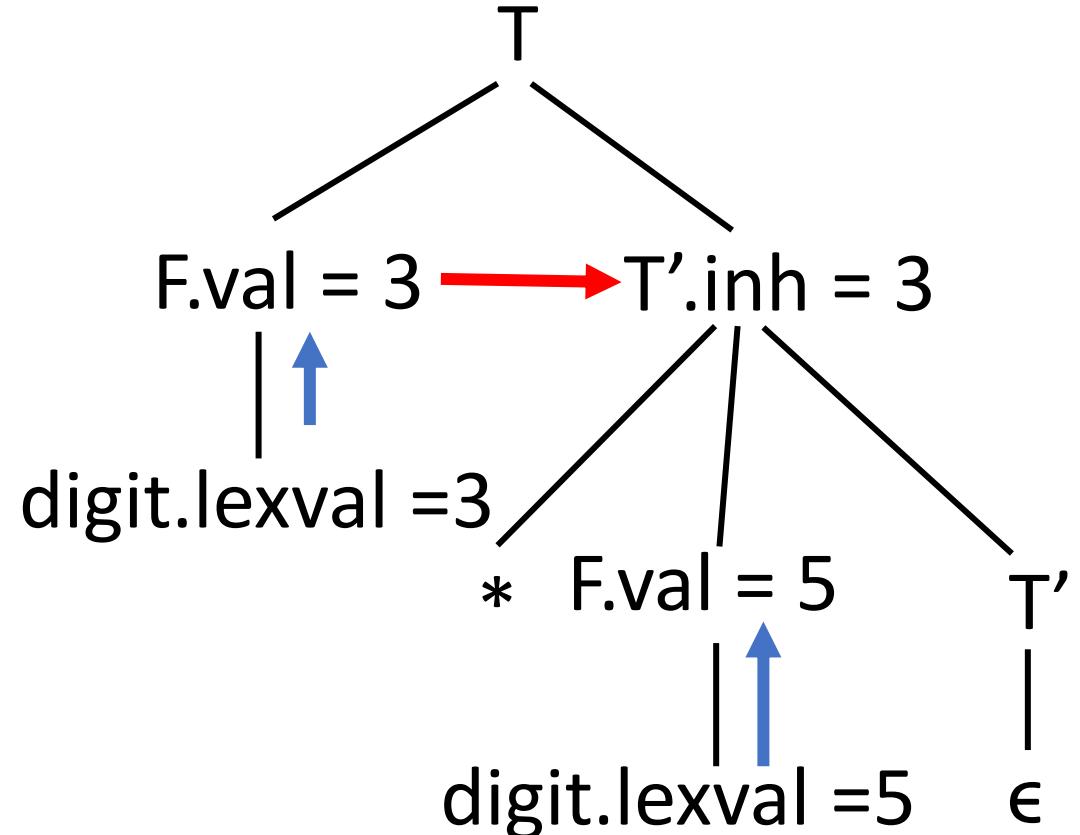


SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

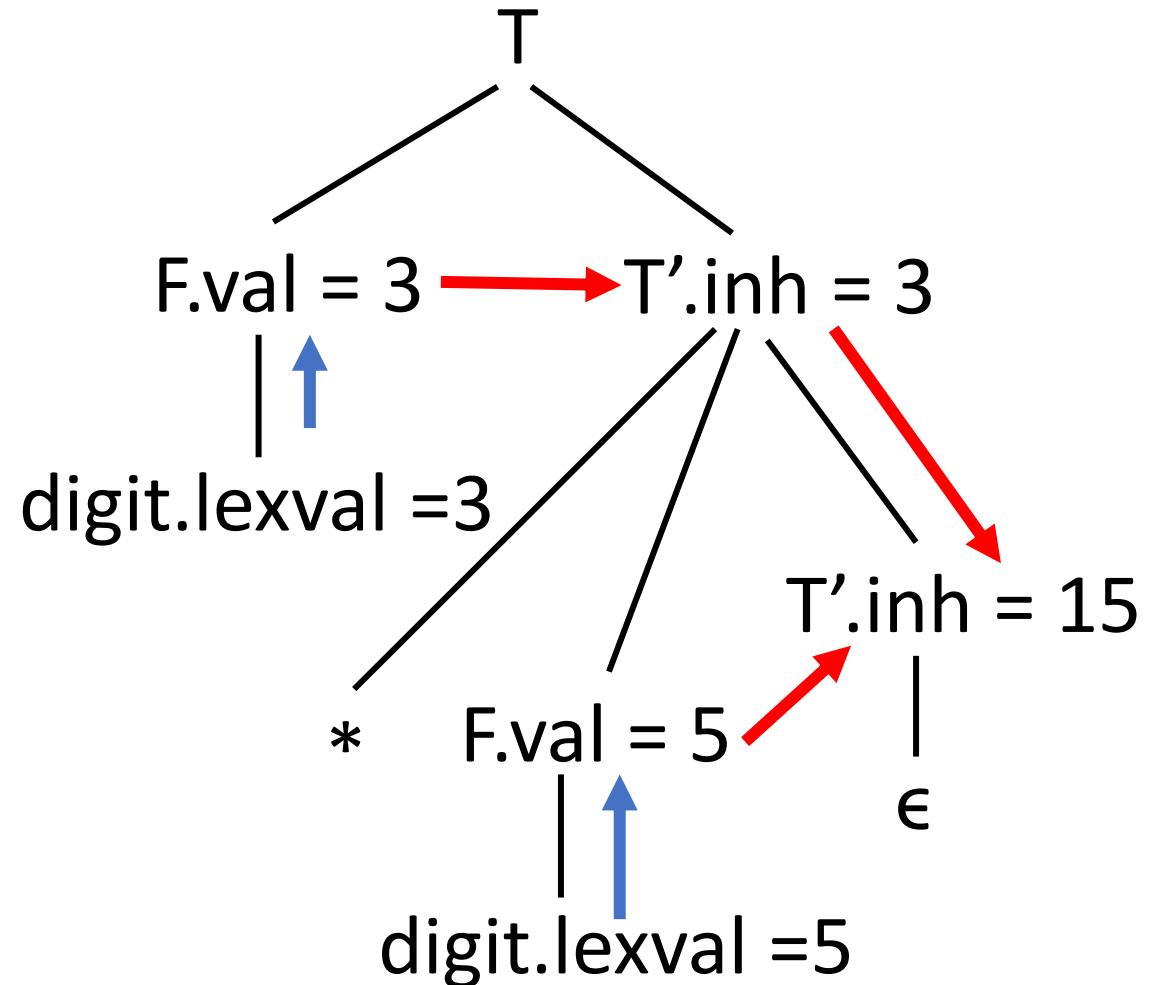


SDD based on a grammar suitable for top-down parsing

## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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



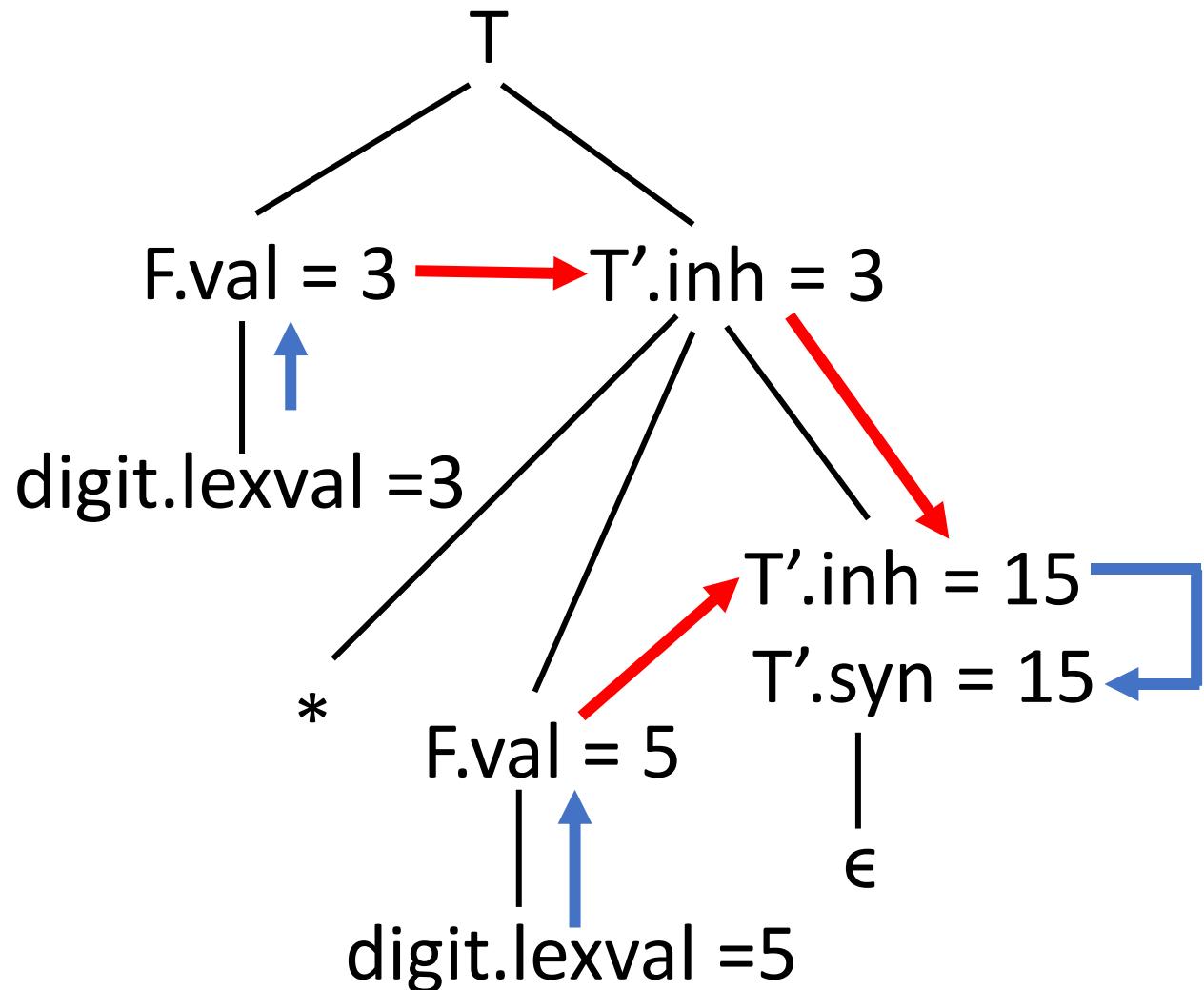
SDD based on a grammar suitable for top-down parsing

# Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

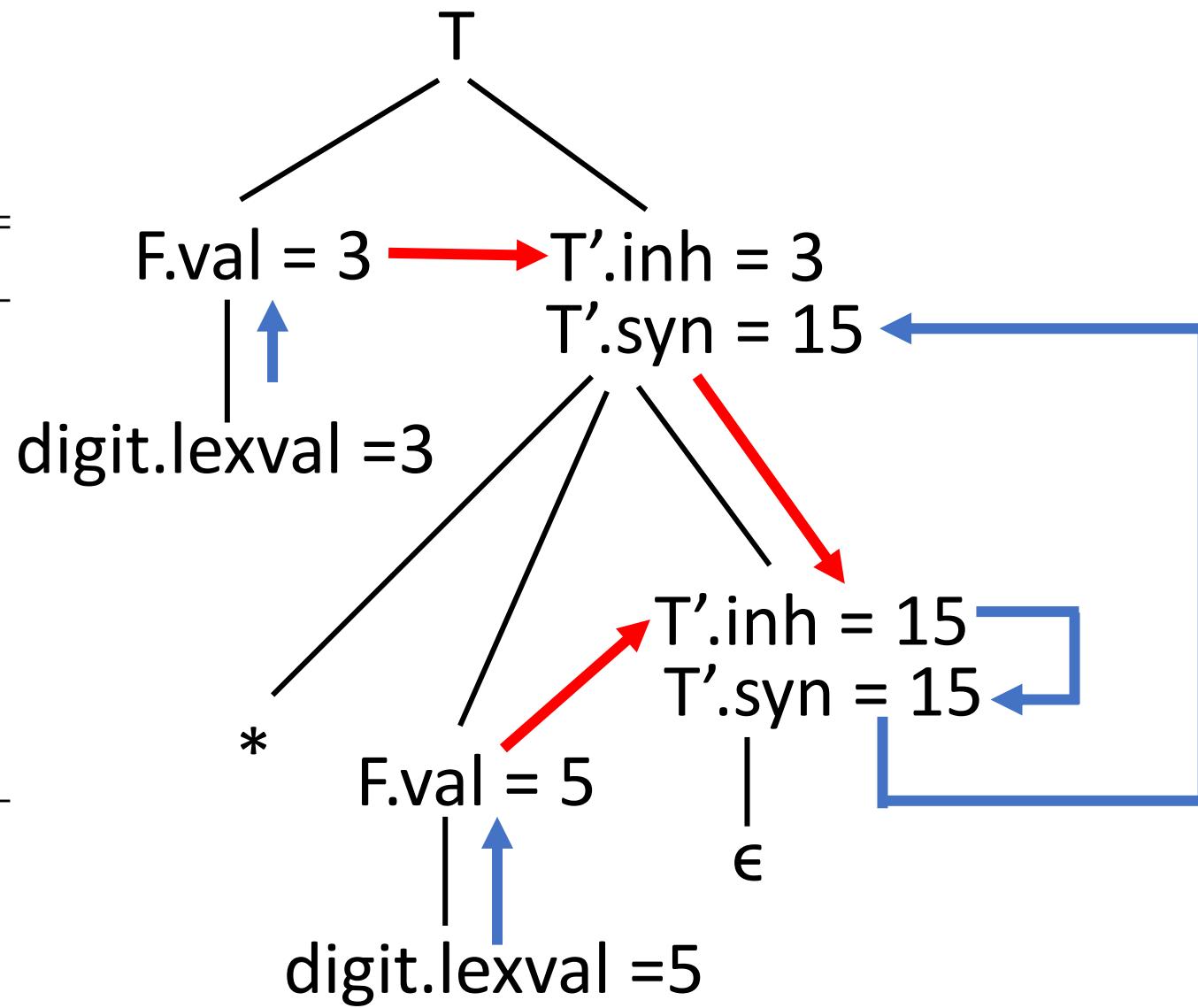


## Example 5.3 (Cont...)

## Annotated Parse Tree for $3 * 5$

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

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

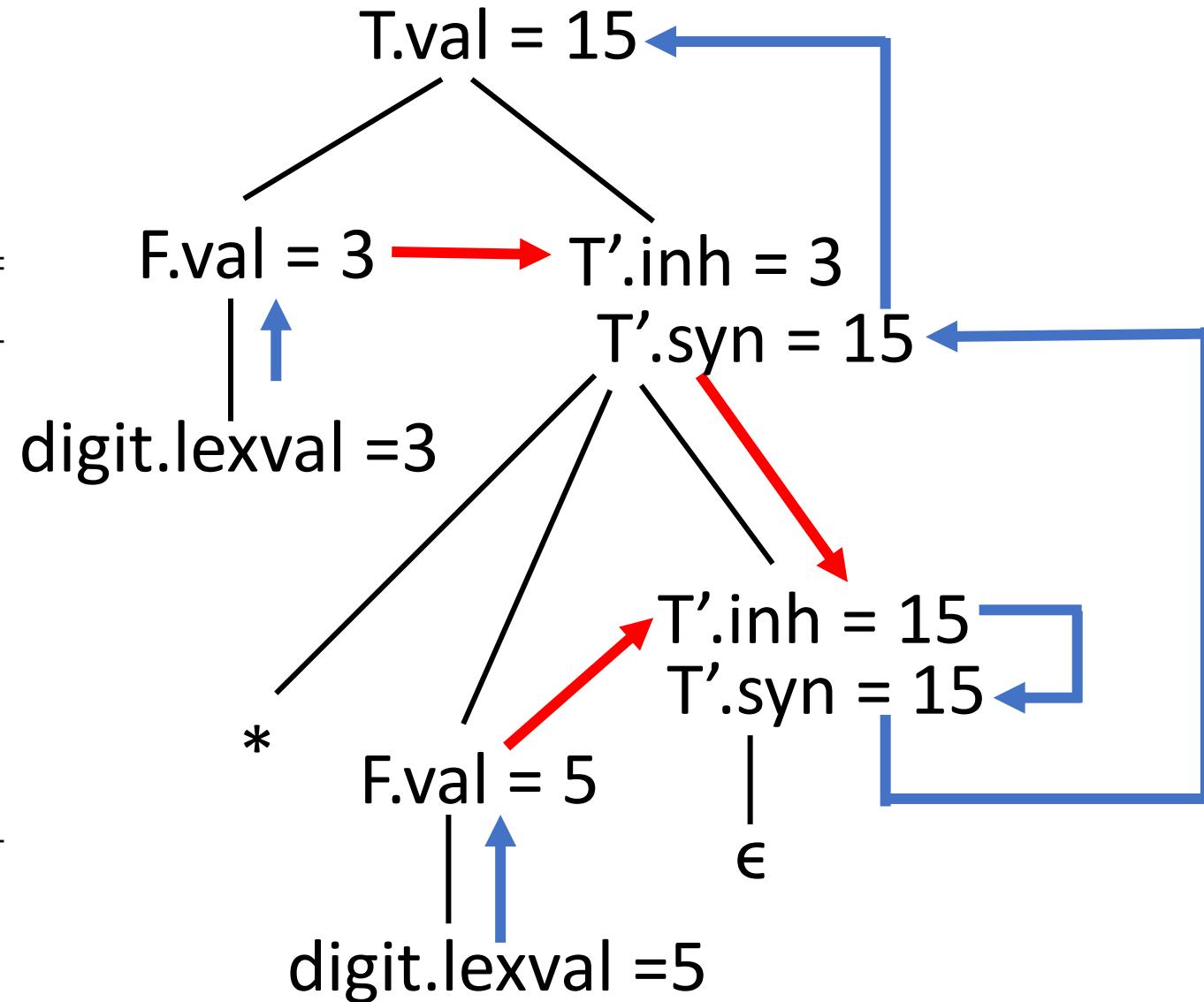


## Example 5.3 (Cont...)

Annotated Parse Tree for  $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing



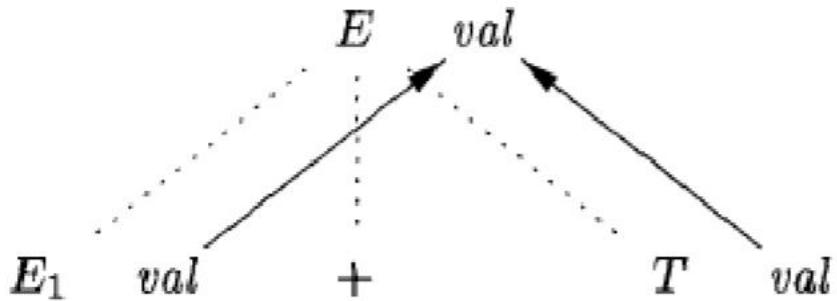
Annotated Parse Tree for  $3 * 5$

# Evaluation Orders for SDD's

# 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  $E_2.val$

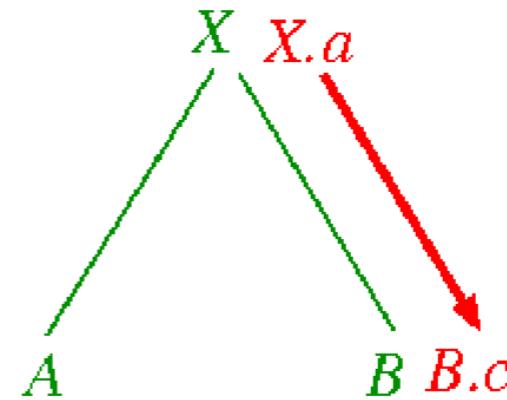
- Suppose that a semantic rule associated with a production  $p: E \rightarrow E_1 + T$  defines the value of synthesized attribute  $E.val$  in terms of the value of  $E_1.val$  and  $T.val$
- Then, the dependency graph has an edge from  $E_1.val$  to  $E.val$  and an edge from  $T.val$  to  $E.val$

# 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...)

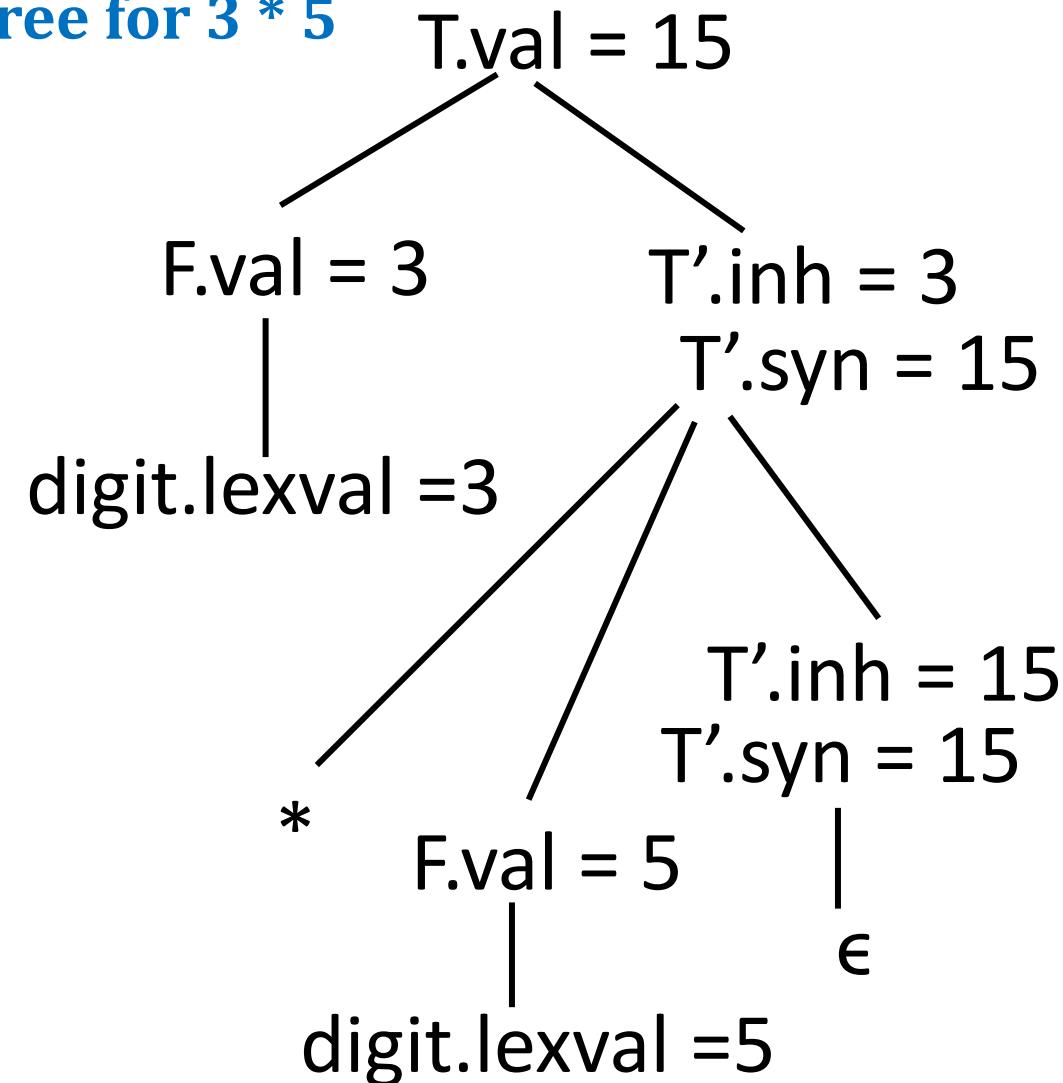
- For each node N labeled B that corresponds to an occurrence of this B in the **body of production p**, create an **edge to attribute c** at N from the attribute a at the node M that corresponds to this occurrence of X
- Note that M could be either the **parent or a sibling** of N

# Example 5.5

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing



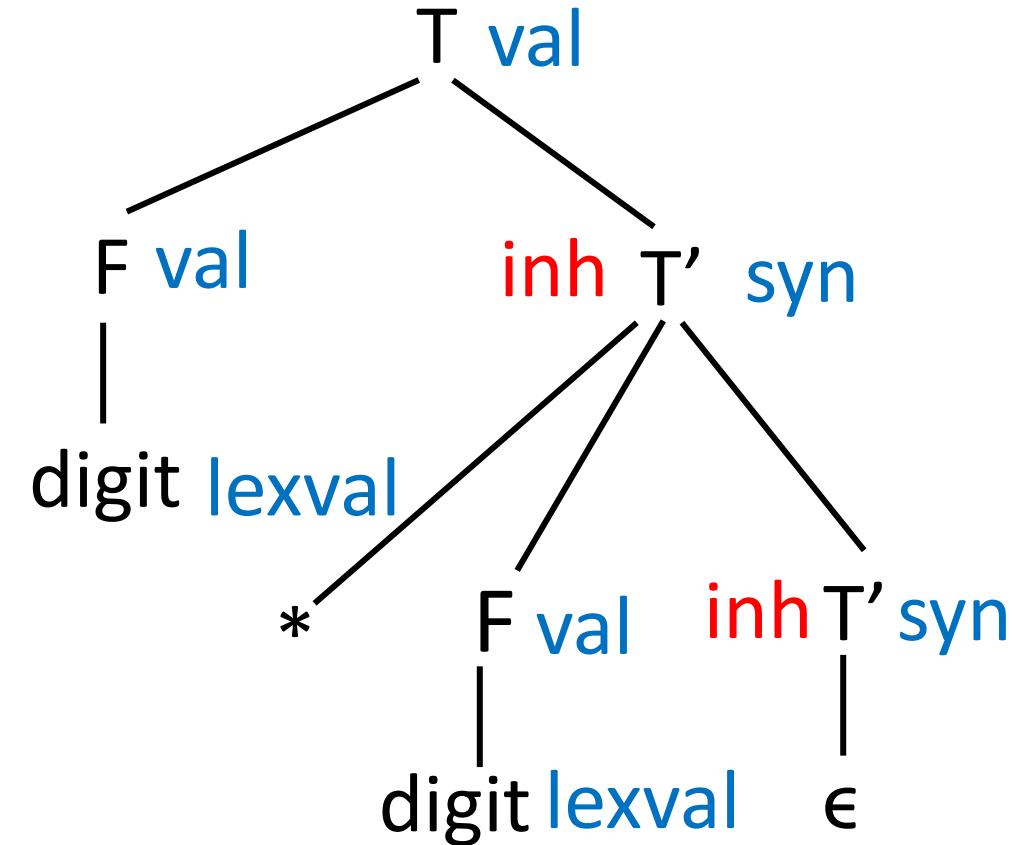
Annotated Parse Tree for  $3 * 5$

# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

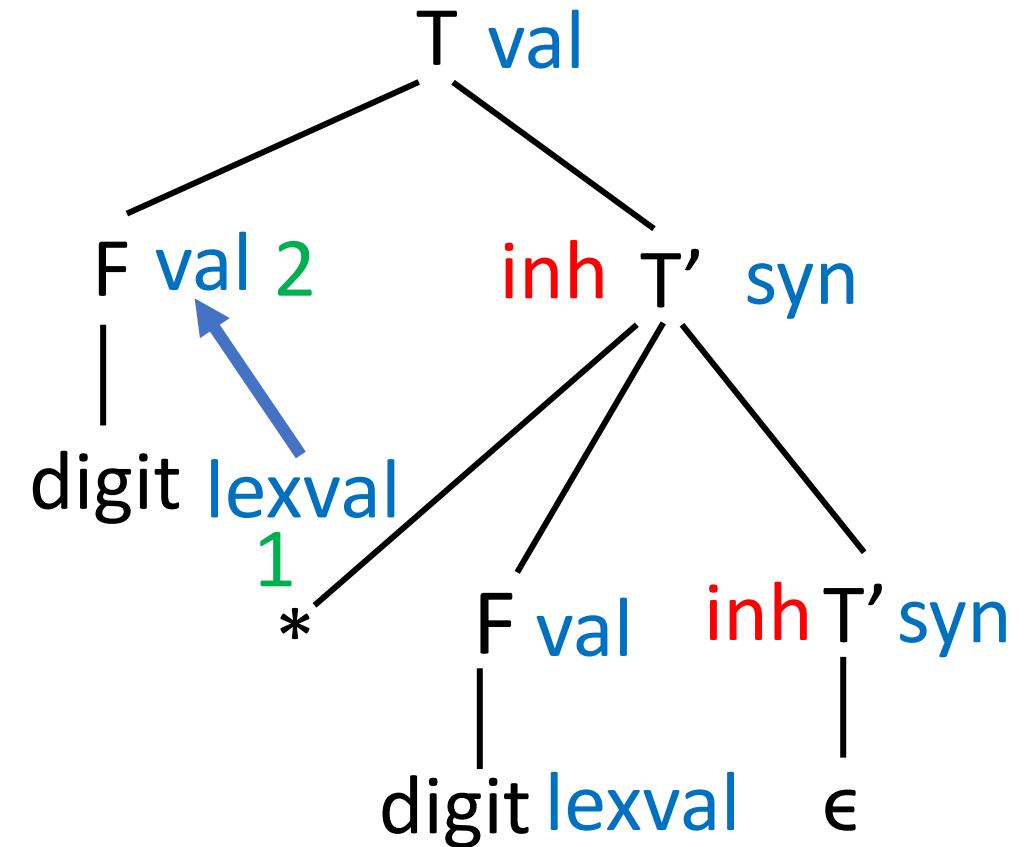


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

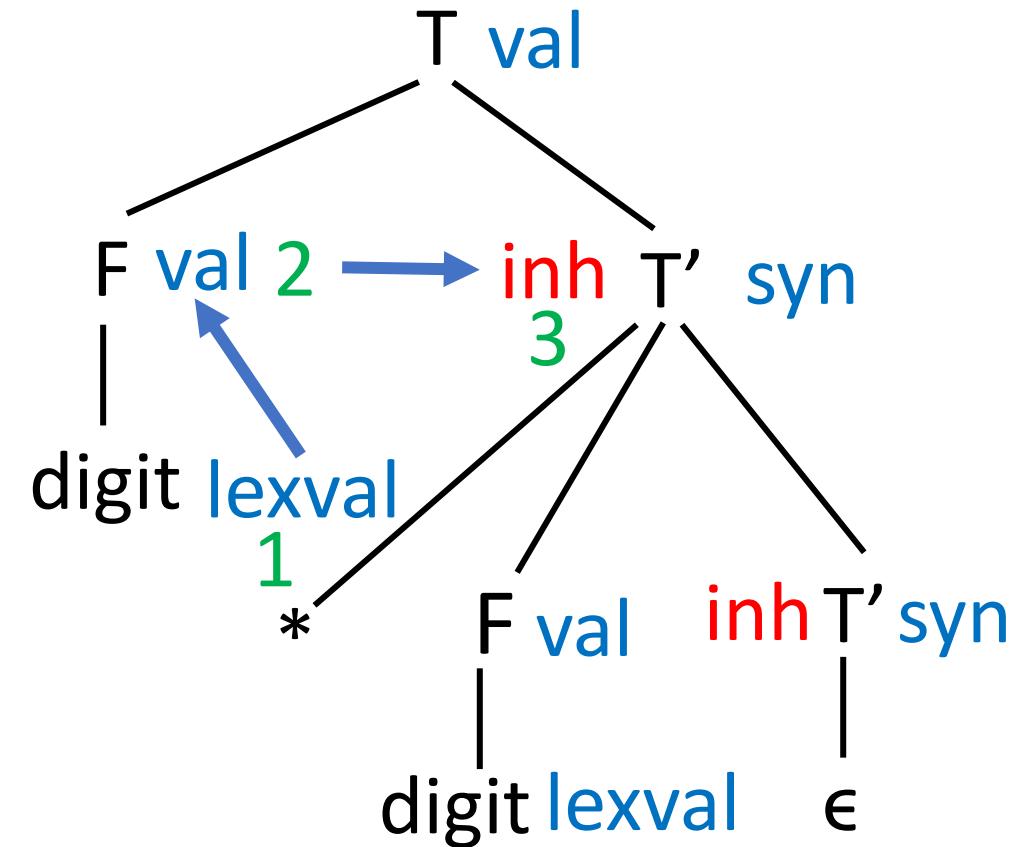


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

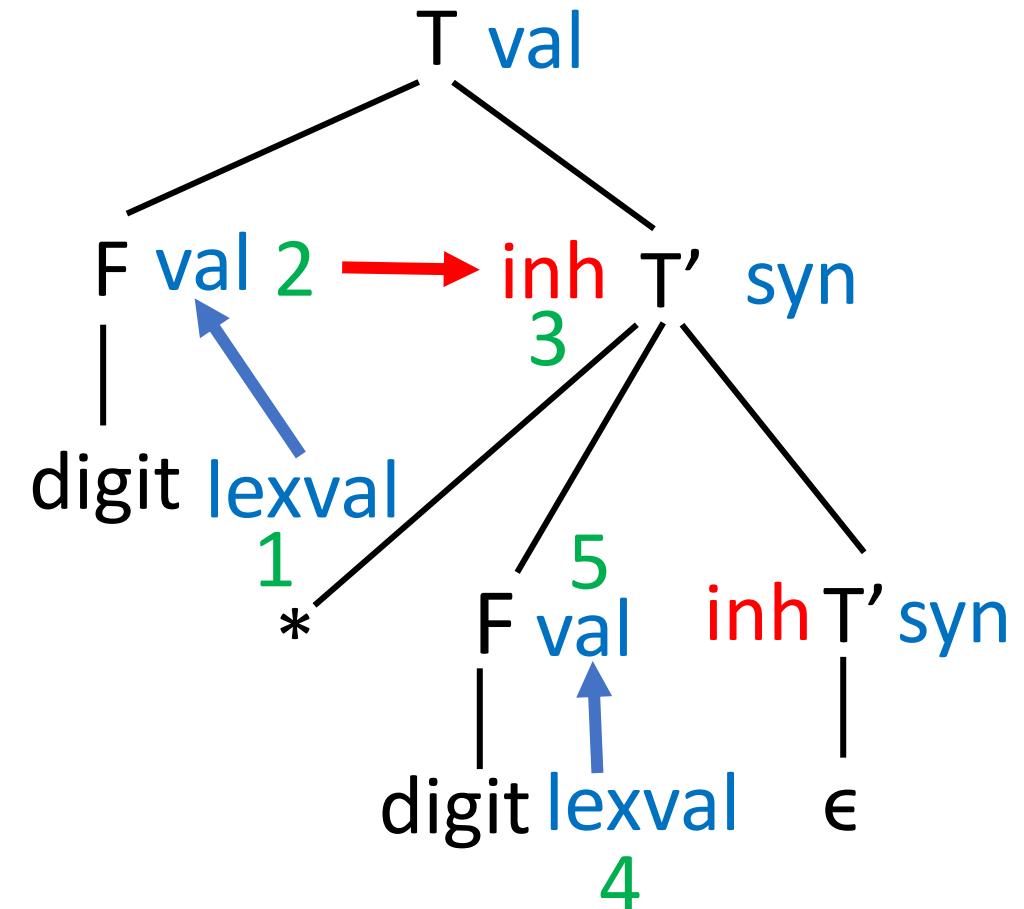


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

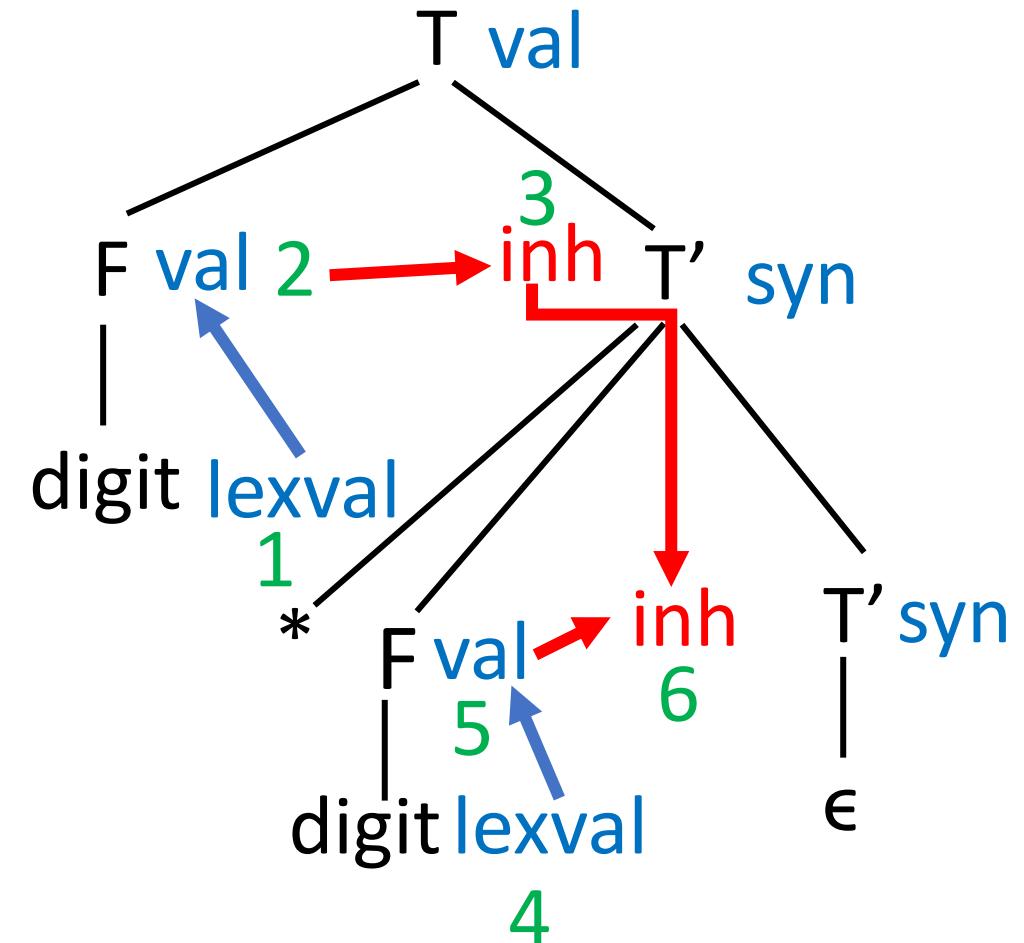


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

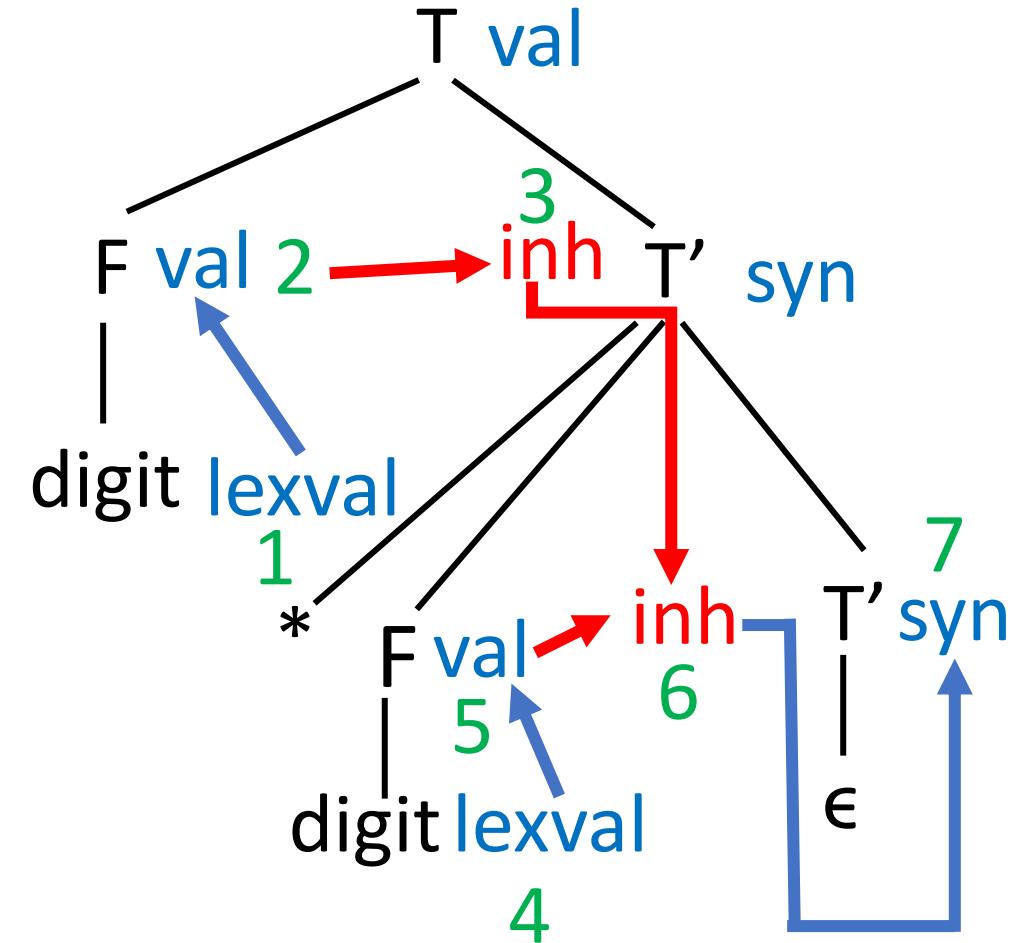


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

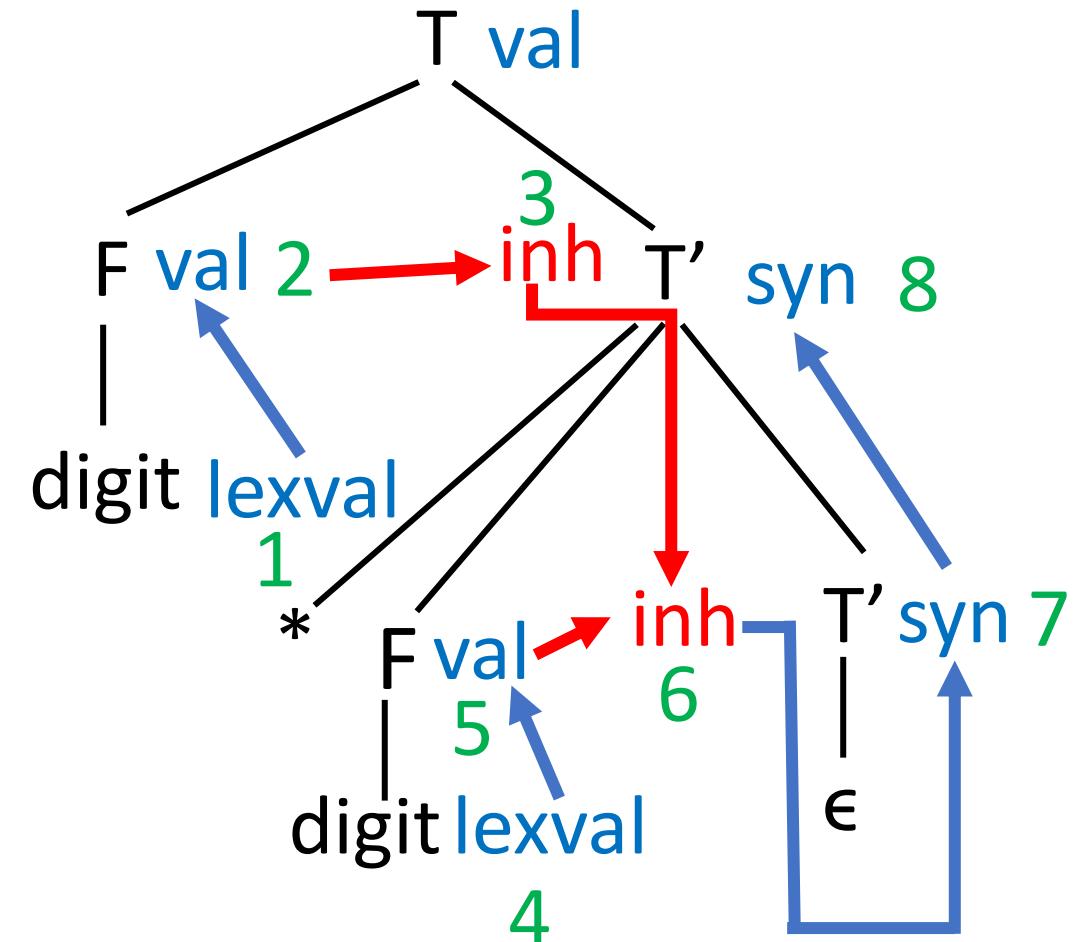


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing

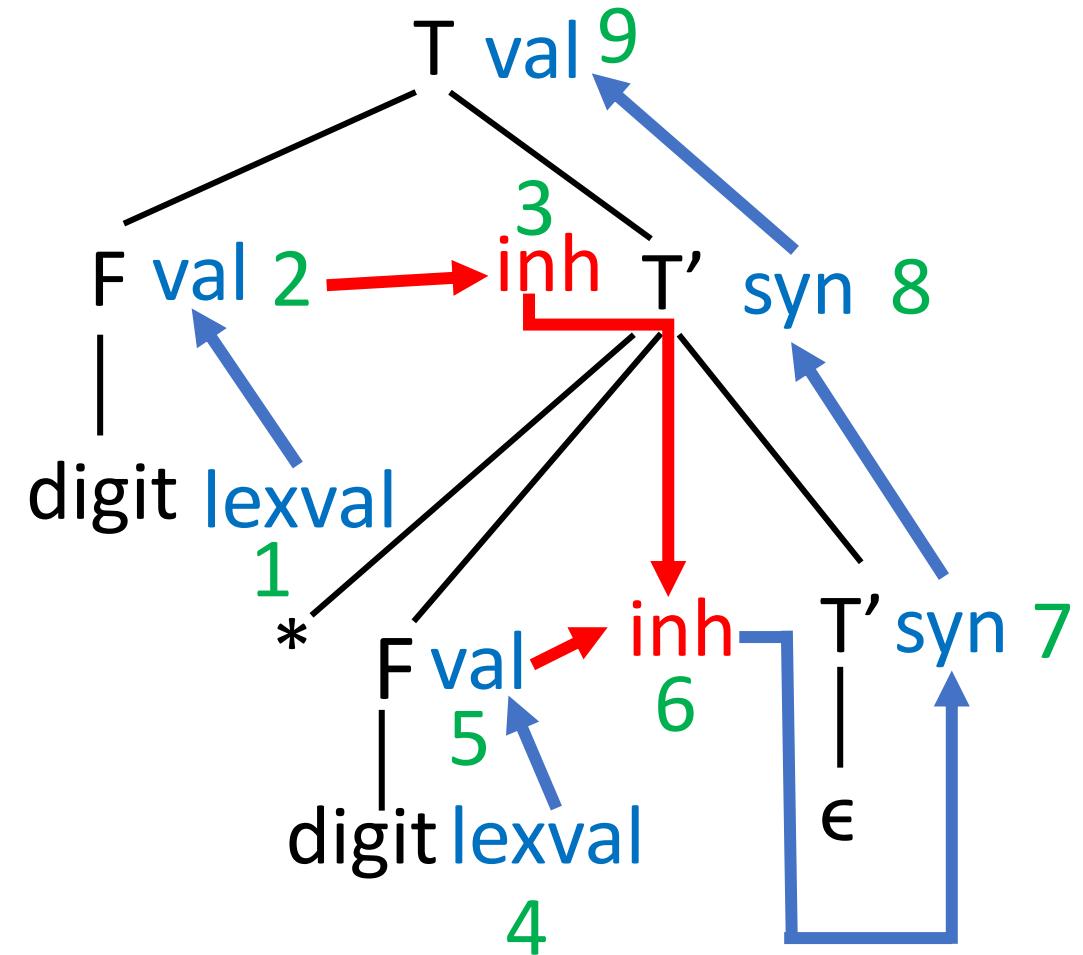


# Example 5.5 (cont..)

## Dependency Graph for the Annotated Parse Tree for $3 * 5$

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

SDD based on a grammar suitable  
for top-down parsing



Dependency Graph for the Annotated Parse Tree

# Ordering the Evaluation of Attributes (Topological Sort)

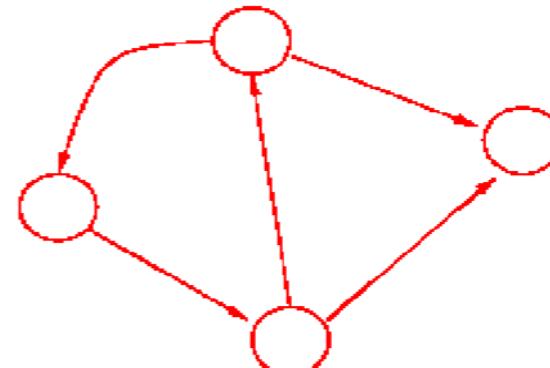
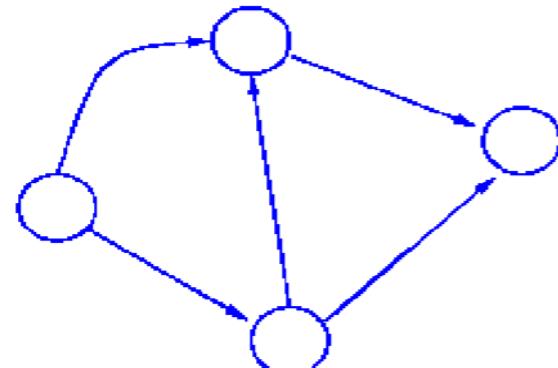
- The dependency graph characterizes the **possible orders** in which we can evaluate the attributes at the various nodes of a parse tree
- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N
- Thus, the only allowable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

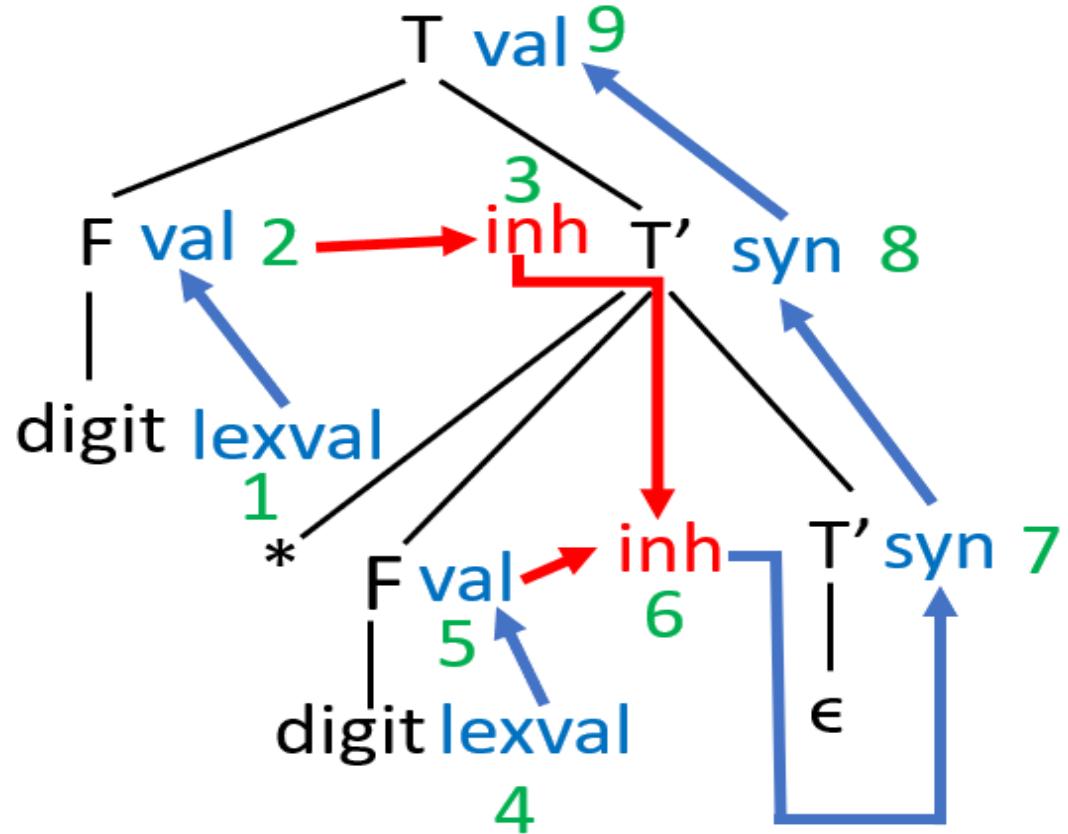
# Ordering the Evaluation of Attributes(Cont...)

- ❑ 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 5.6

## Topological Sort of the Dependency Graph for $3 * 5$

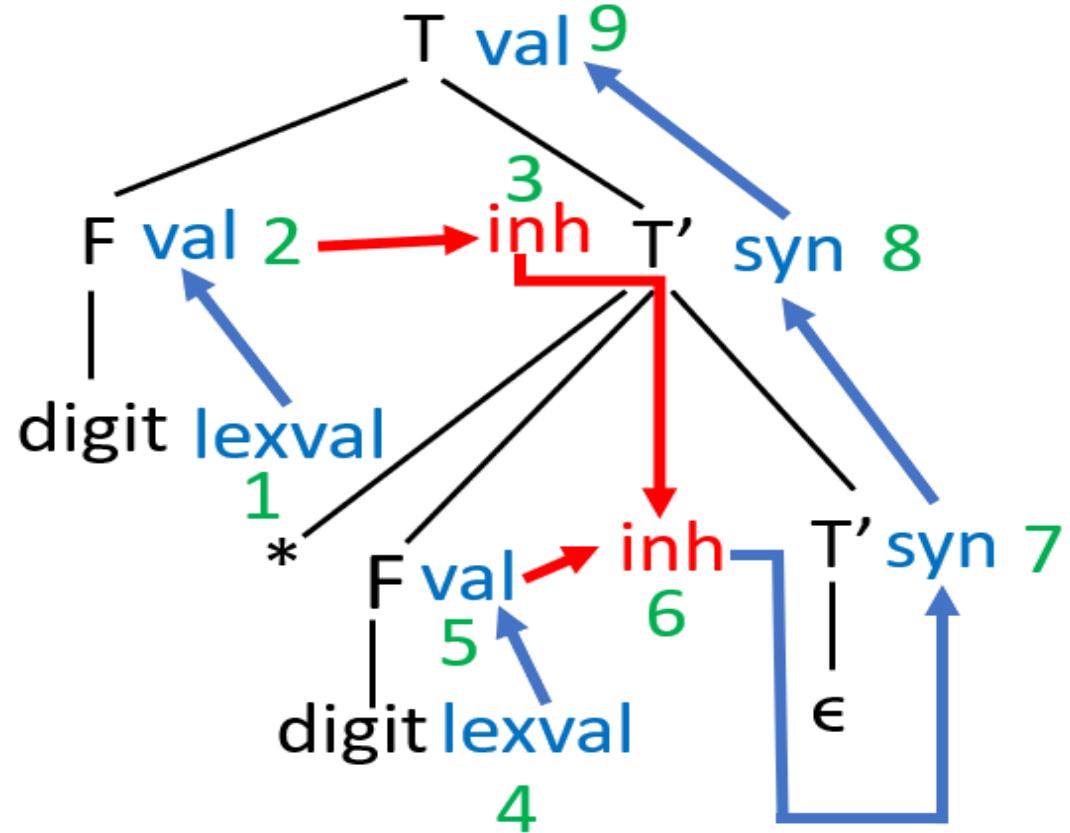


- The dependency graph of figure has no cycles
- One topological sort is the order in which the nodes have already been numbered: 1,2,3,4,5,6,7,8,9

Dependency Graph for the Annotated Parse Tree

## Example 5.6 (Cont...)

### Topological Sort of the Dependency Graph for $3 * 5$



- 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 1,2,4,5,3,6,7,8,9

Dependency Graph for the Annotated Parse Tree

# 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

- An SDD is S-attributed if **every attribute is synthesized**
- The SDD of the 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) $L \rightarrow E \text{ n}$	$L.\text{val} = E.\text{val}$
2) $E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
3) $E \rightarrow T$	$E.\text{val} = T.\text{val}$
4) $T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$
5) $T \rightarrow F$	$T.\text{val} = F.\text{val}$
6) $F \rightarrow ( E )$	$F.\text{val} = E.\text{val}$
7) $F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.lexval$

Syntax-directed definition of a simple desk calculator

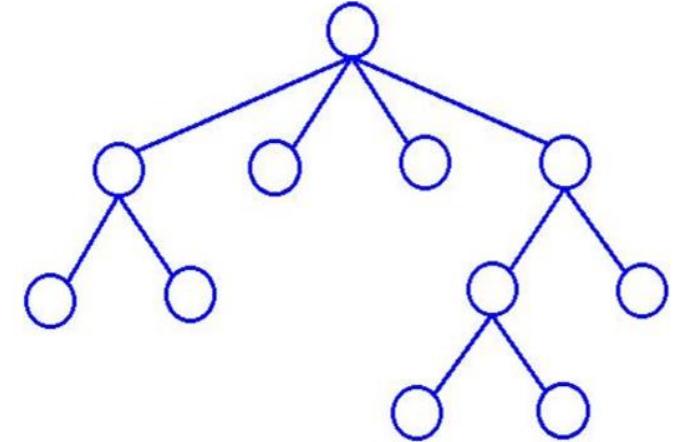
# S-Attributed Definitions(Cont...)

- 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  $A \rightarrow X_1 X_2 \dots X_n$ , 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  $X_1, X_2, \dots, X_{i-1}$  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)	$L \rightarrow E \text{ n}$	$L.val = E.val$
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)	$F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

Syntax-directed definition of a simple desk calculator

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

# Example

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

Syntax-directed definition of a simple desk calculator

- Is this SDD L-Attributed?
  - Yes. Because all S-attributed SDD are L-attributed SDD because they both support synthesized attributes.

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

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

- Is this SDD S-Attributed?
  - **No.** Because grammar symbol  $T'$  has an inherited attribute.
- 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 defined
- So **the SDD is neither S-attributed or L-attributed**

# **Semantic Rules with Controlled Side Effects**

# Semantic Rules with Controlled Side Effects

- In practice, translations involve side effects:
  - a desk calculator might **print** a result
  - a semantic analyzer 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) $L \rightarrow E \text{ n}$	$L.\text{val} = E.\text{val}$
2) $E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
3) $E \rightarrow T$	$E.\text{val} = T.\text{val}$
4) $T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$
5) $T \rightarrow F$	$T.\text{val} = F.\text{val}$
6) $F \rightarrow ( E )$	$F.\text{val} = E.\text{val}$
7) $F \rightarrow \text{digit}$	$F.\text{val} = \text{digit.lexval}$

- 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) $L \rightarrow E \text{n}$	$\text{print}(E.\text{val})$

Syntax-directed definition of a simple desk calculator

# Semantic Rules with Controlled Side Effects (Cont..)

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

- Semantic rules that are executed for their side effects, such as  $\text{print}(E.\text{val})$ , 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  $\text{print}$  statement is executed at the end, after the result is computed into  $E.\text{val}$

## Example 5.10

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) $L \rightarrow \text{id}$	$\text{addType}(\text{id}.entry, L.inh)$

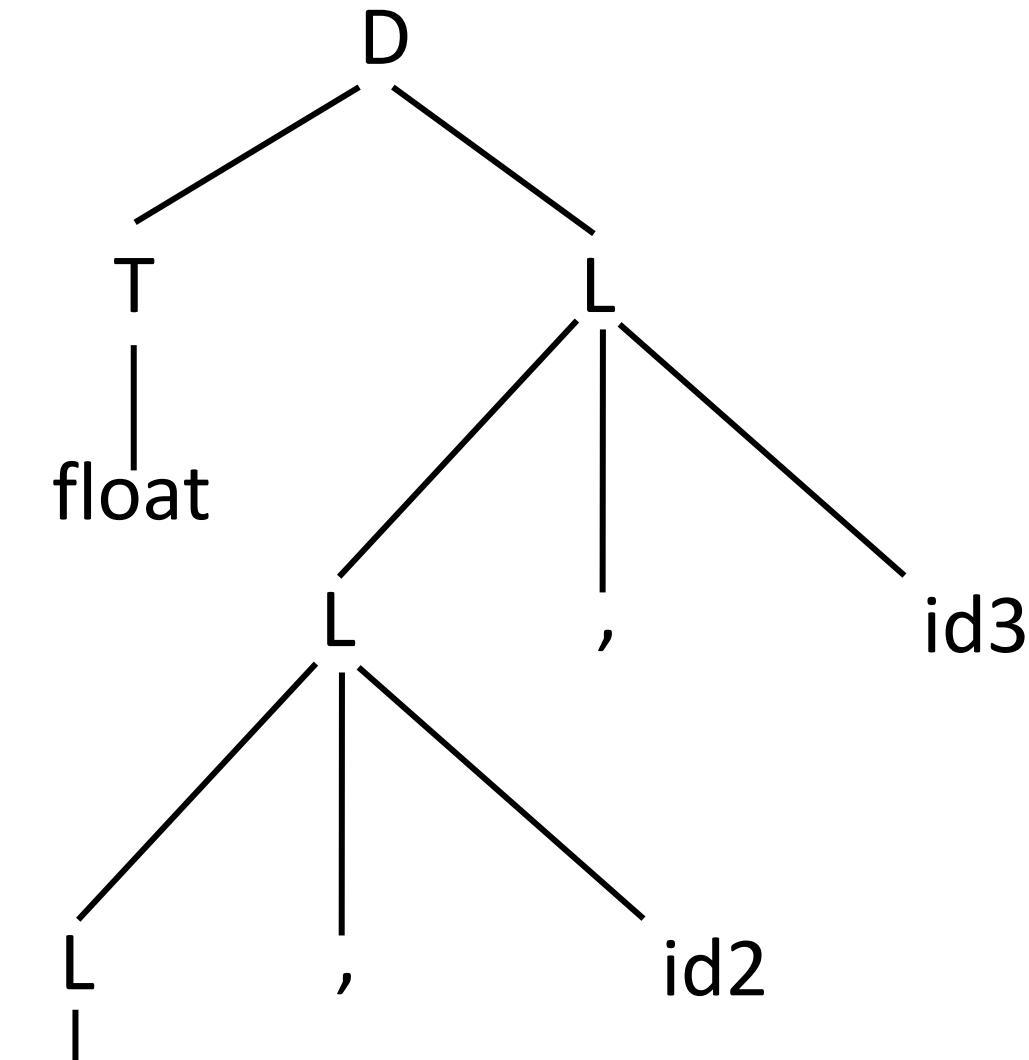
Syntax-directed definition for simple type declarations

## Example 5.10 (Cont...)

Dependency Graph 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) $L \rightarrow \text{id}$	$\text{addType}(\text{id.entry}, L.inh)$

Syntax-directed definition for simple type declarations



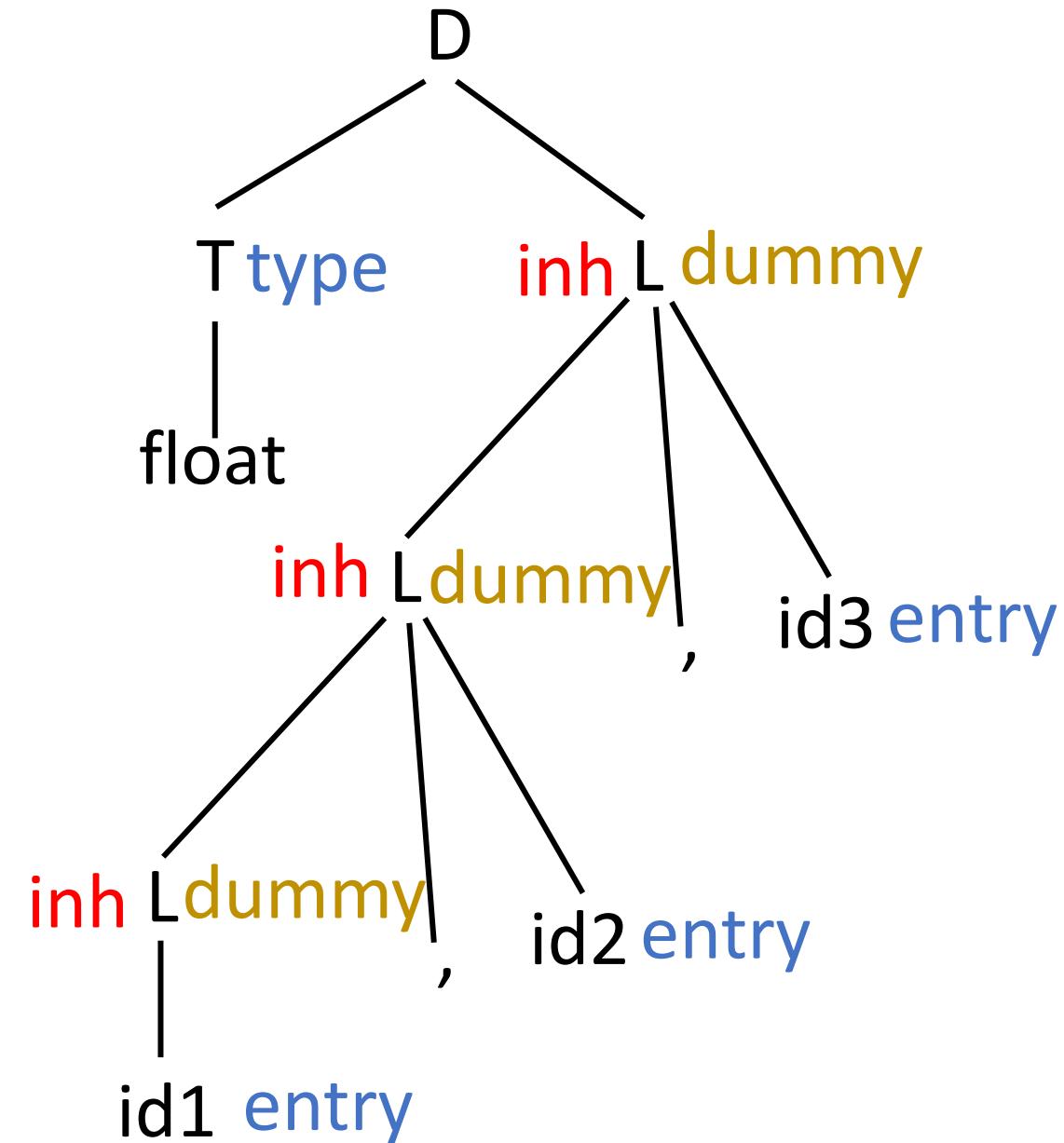
Parse Tree for float id1, id2, id3

## Example 5.10 (Cont...)

Dependency Graph 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) $L \rightarrow \text{id}$	$\text{addType}(\text{id.entry}, L.inh)$

Syntax-directed definition for simple type declarations

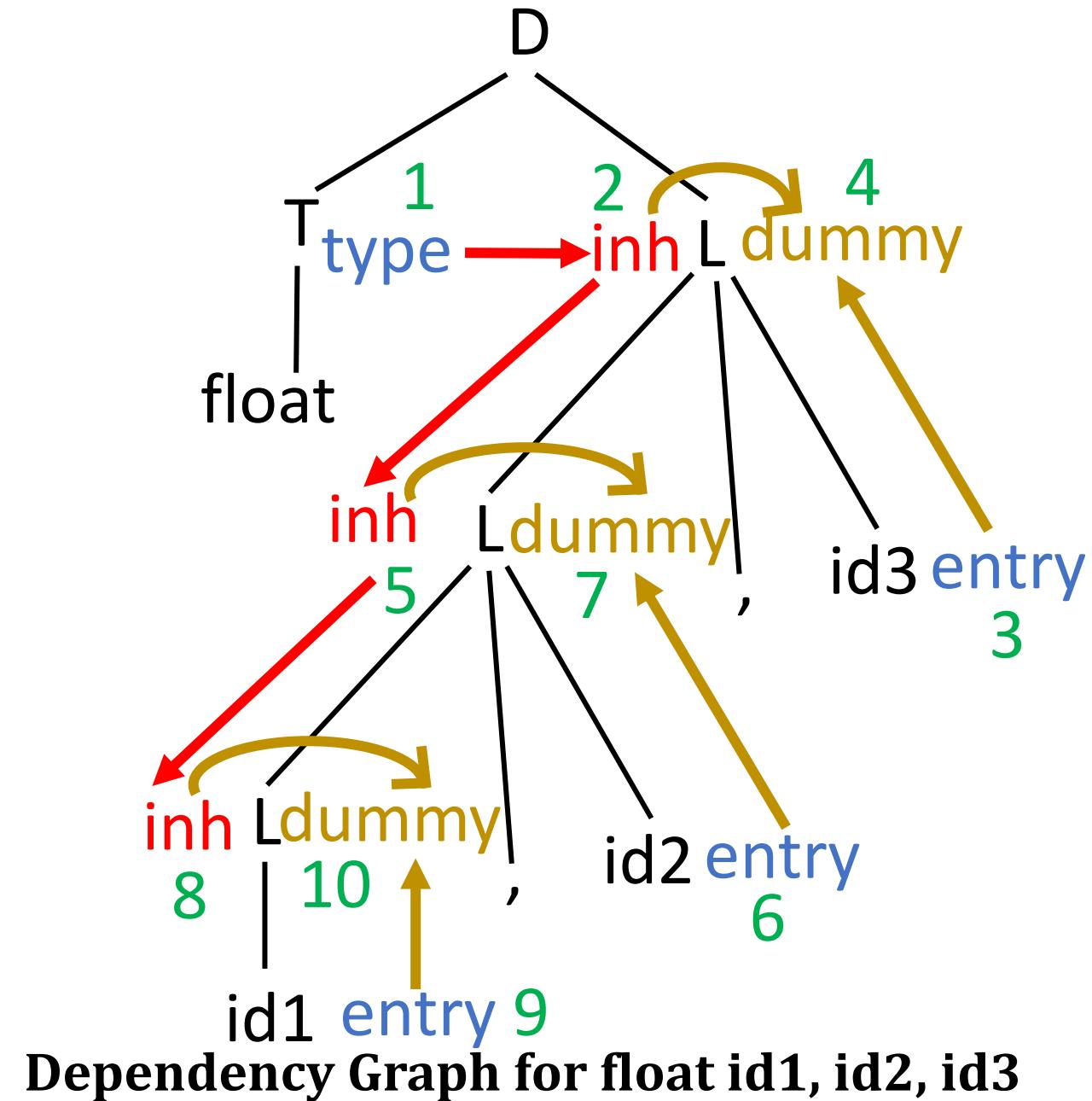


## Example 5.10 (Cont...)

### Dependency Graph 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) $L \rightarrow \text{id}$	$\text{addType}(\text{id.entry}, L.inh)$

Syntax-directed definition for simple type declarations



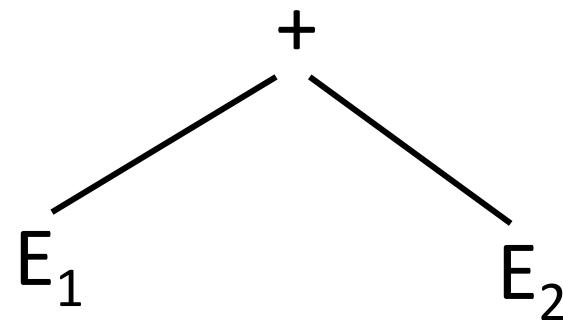
# Applications of Syntax-Directed Translation

# 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

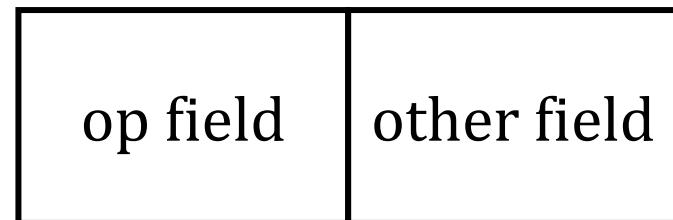
- Each node in a syntax tree represents a construct
- The children of the node represent the meaningful components of the construct
- A syntax-tree node representing an expression  $E_1 + E_2$  has label  $+$  and two children representing the subexpressions  $E_1$  and  $E_2$



Syntax Tree Node for an Expression  $E_1+E_2$

# Construction of Syntax Trees (Cont...)

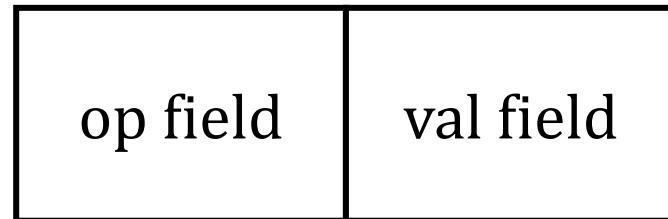
- We shall implement the nodes of a syntax tree by objects with a suitable number of fields
- Each object will have an op field that is the label of the node
- The objects will have additional fields



**Syntax Tree Node Representation as a Record**

# 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



**Syntax Tree Node for a Leaf Node**

# 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, c<sub>1</sub>, c<sub>2</sub>,..., c<sub>k</sub>)** creates an object with first field op and k additional fields for the k children c<sub>1</sub>,..., c<sub>k</sub>



**Syntax Tree Node for an Interior Node with two children**

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

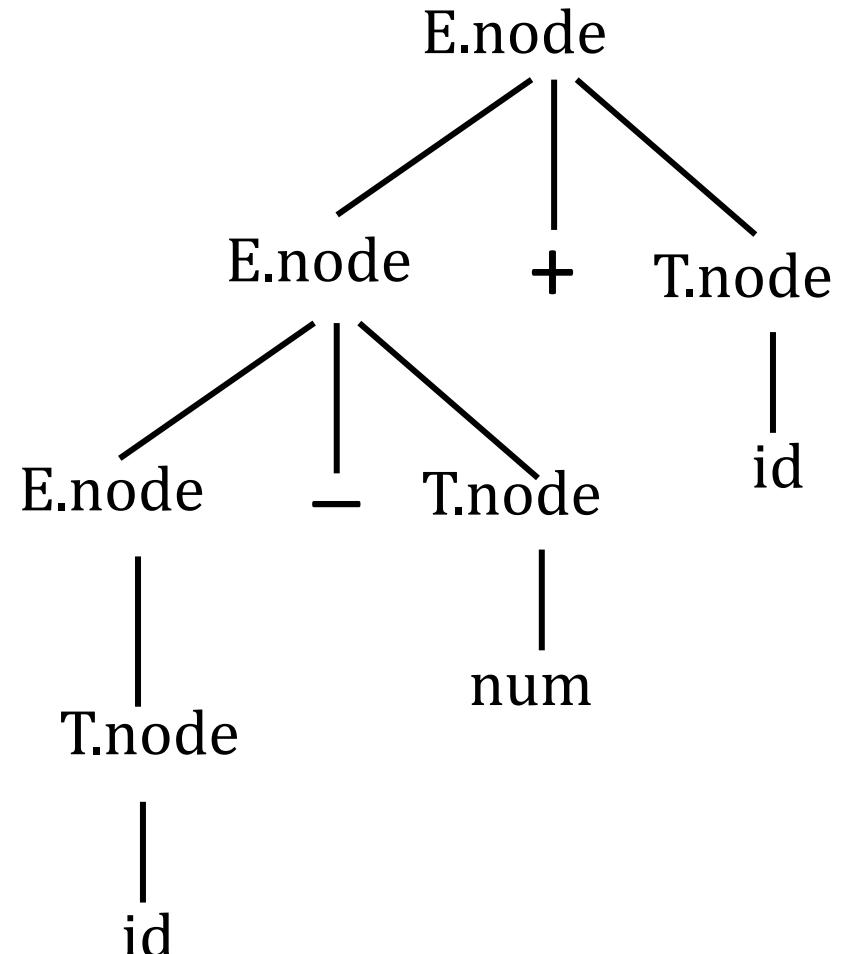
Constructing syntax trees for simple expressions

## Example 5.11 (Cont...)

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) $T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.val)$

Constructing syntax trees for simple expressions



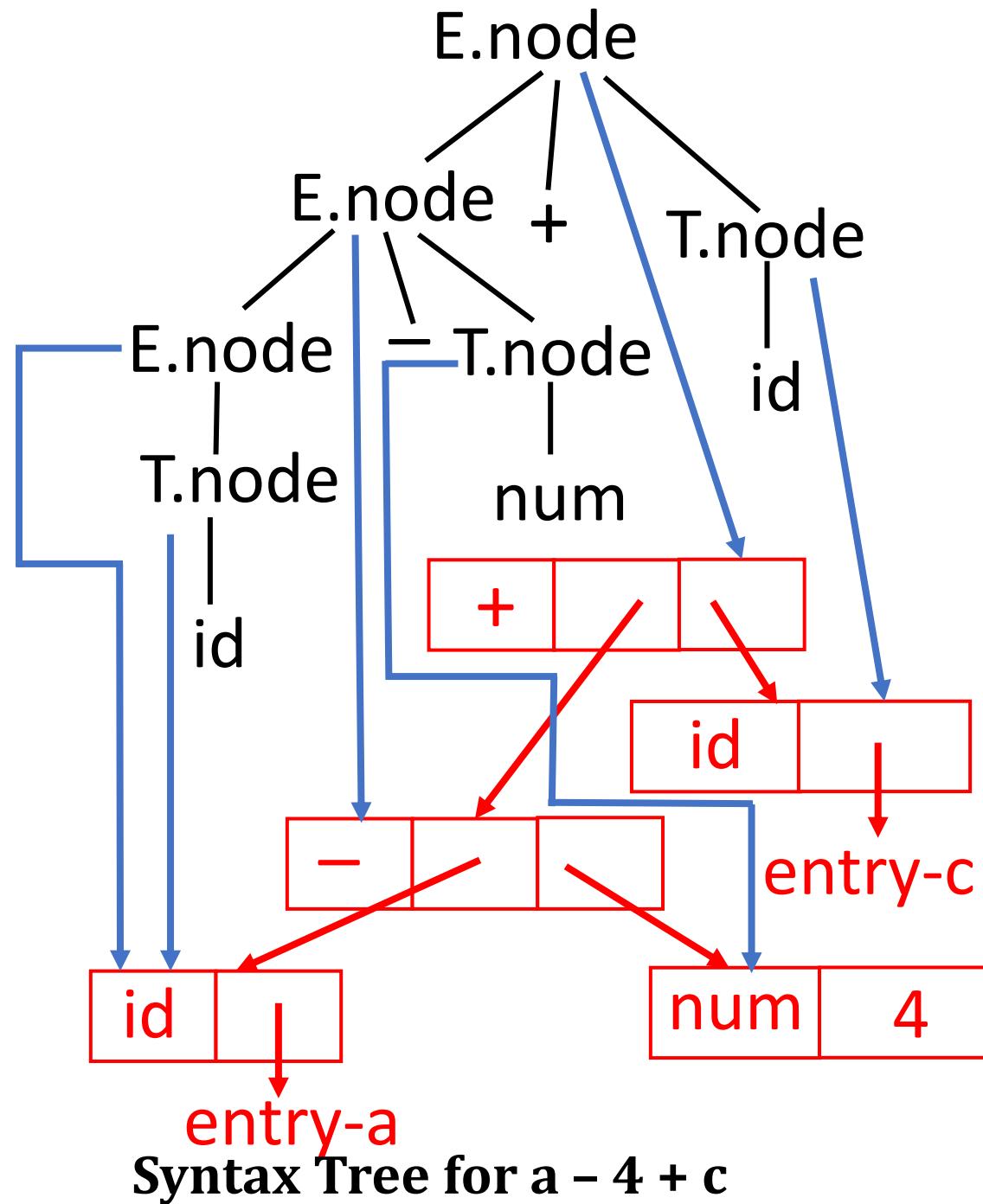
Parse Tree for  $a - 4 + c$

# Example 5.11(Cont...)

Syntax Tree for  $a - 4 + c$

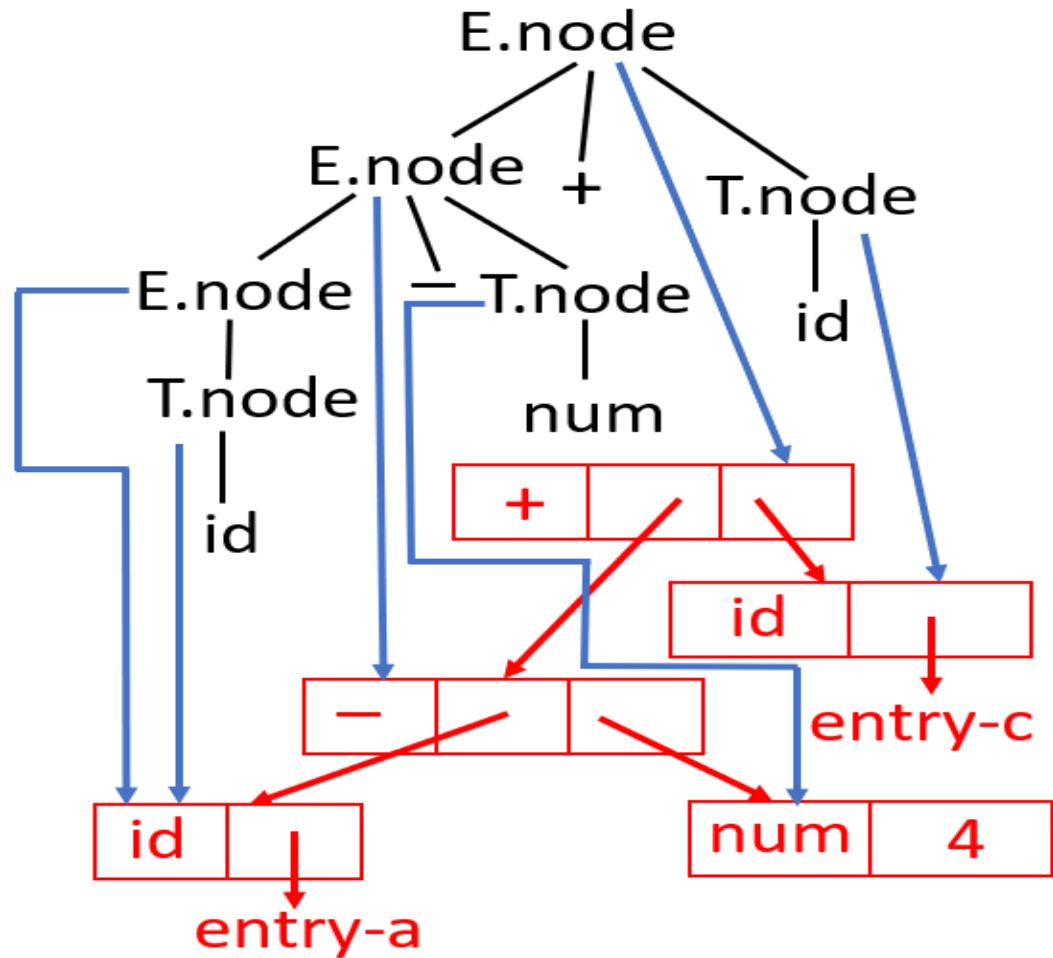
PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+', E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow ( E )$	$T.\text{node}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

Constructing syntax trees for simple expressions



# Example 5.11 (Cont...)

Syntax Tree for  $a - 4 + c$



**Syntax Tree for  $a - 4 + c$**

- 1) `p1 = new Leaf(id, entry-a);`
- 2) `p2 = new Leaf(num, 4);`
- 3) `p3 = new Node('−', p1, p2);`
- 4) `p4 = new Leaf(id, entry-c);`
- 5) `p5 = new Node('+' , p3, p4);`

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) $E \rightarrow T E'$	$E.\text{node} = E'.\text{syn}$ $E'.\text{inh} = T.\text{node}$
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) $T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.val)$

Constructing syntax trees during top-down parsing

# Comparison of Example 5.3 and Example 5.12

PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $F.val = T'.syn$
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) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

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

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

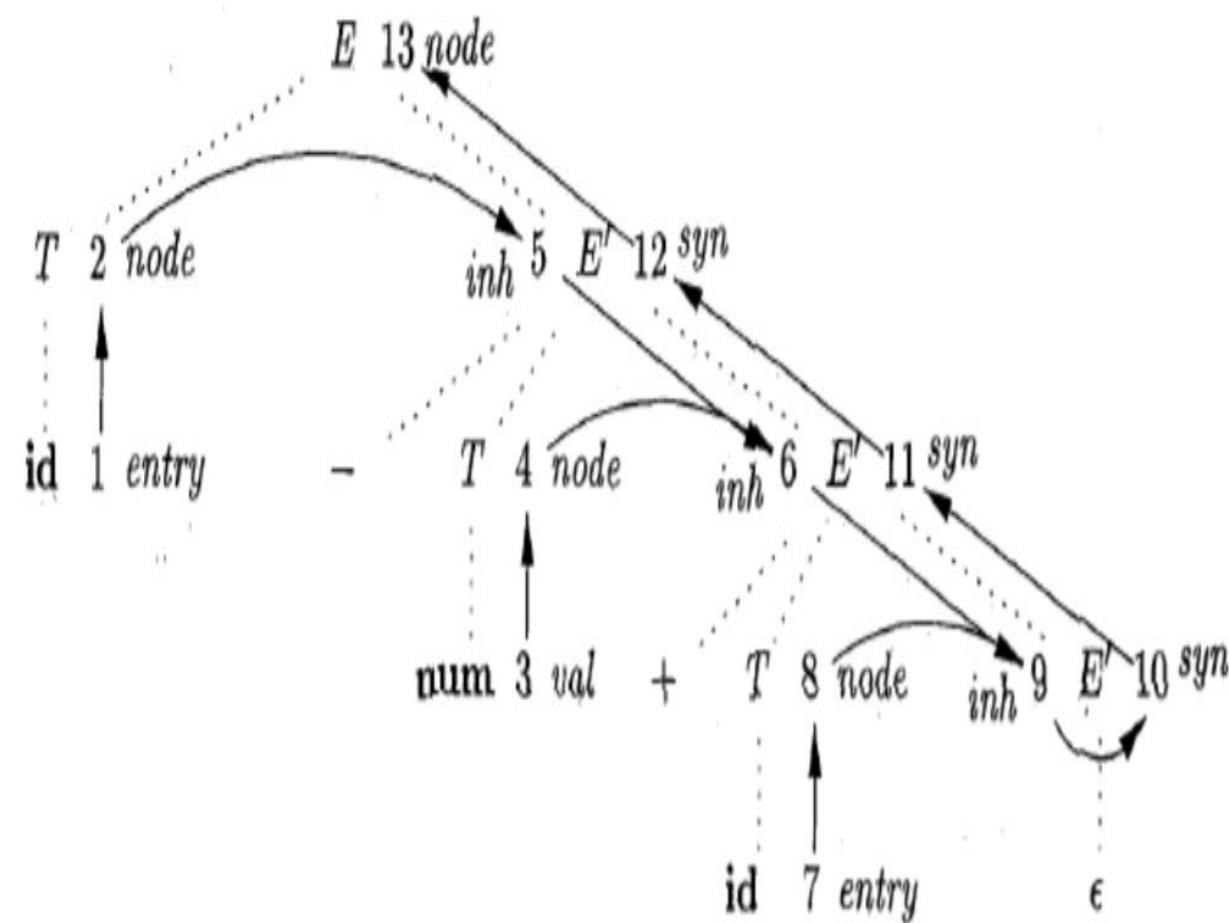
Constructing syntax trees during top-down parsing

# Example 5.12 (Cont...)

## Dependency Graph for a - 4 + c

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow T E'$	$E.\text{node} = E'.\text{syn}$ $E'.\text{inh} = T.\text{node}$
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) $T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf}(\text{num}, \text{num}.val)$

Constructing syntax trees during top-down parsing

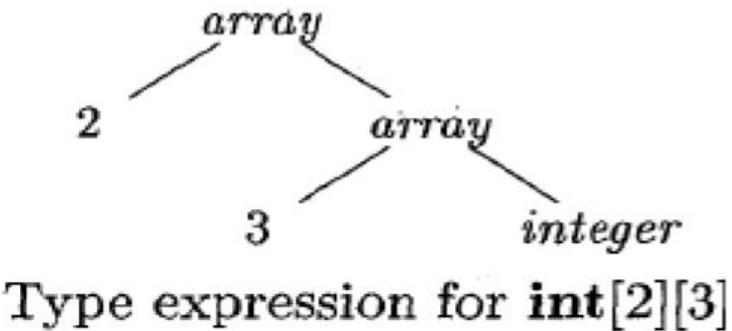


Dependency Graph for a - 4 + c

# Example 5.13

## Structure of Array Types

- 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



- 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...)

### Syntax Directed Definition of Array Types

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

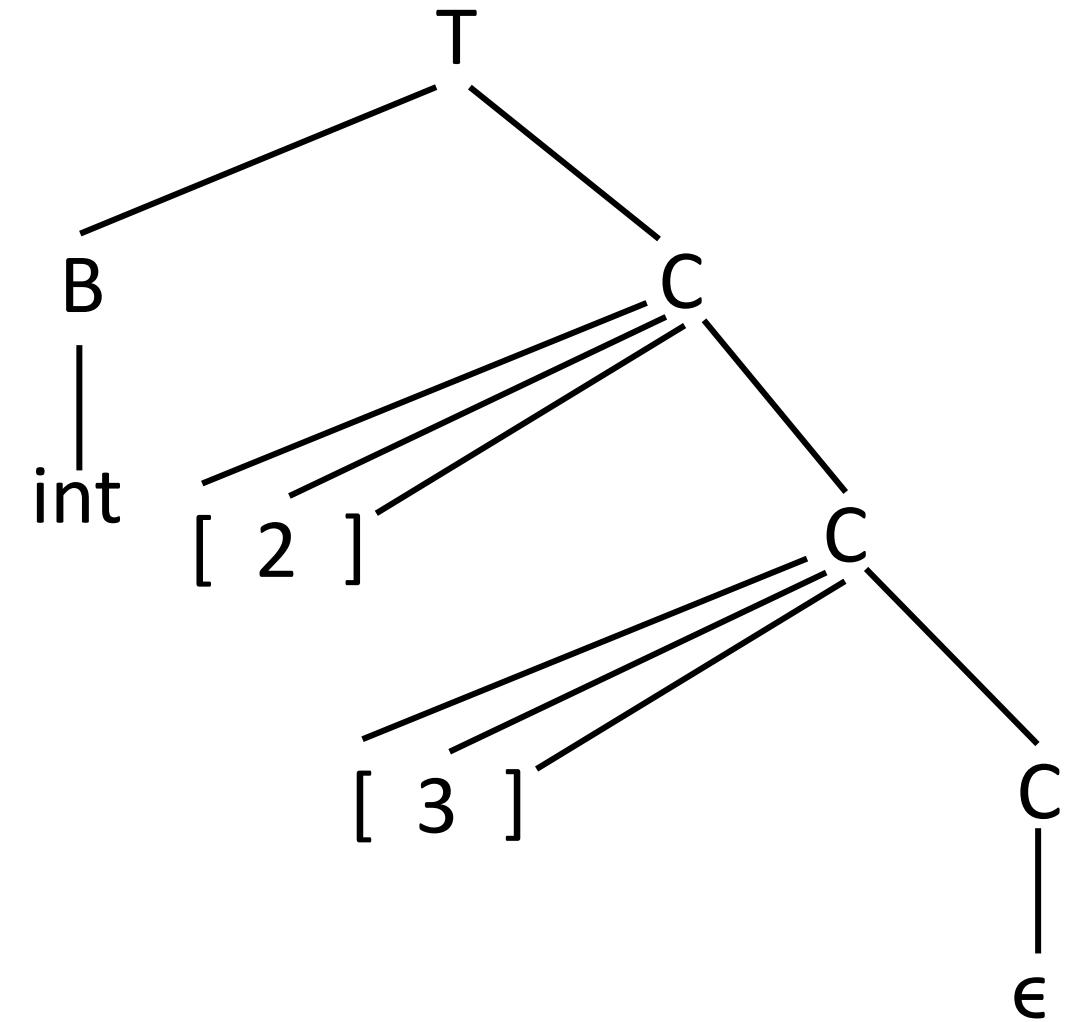
$T$  generates either a basic type or an array type

## Example 5.13 (Cont...)

### Annotated Parse Tree for $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type



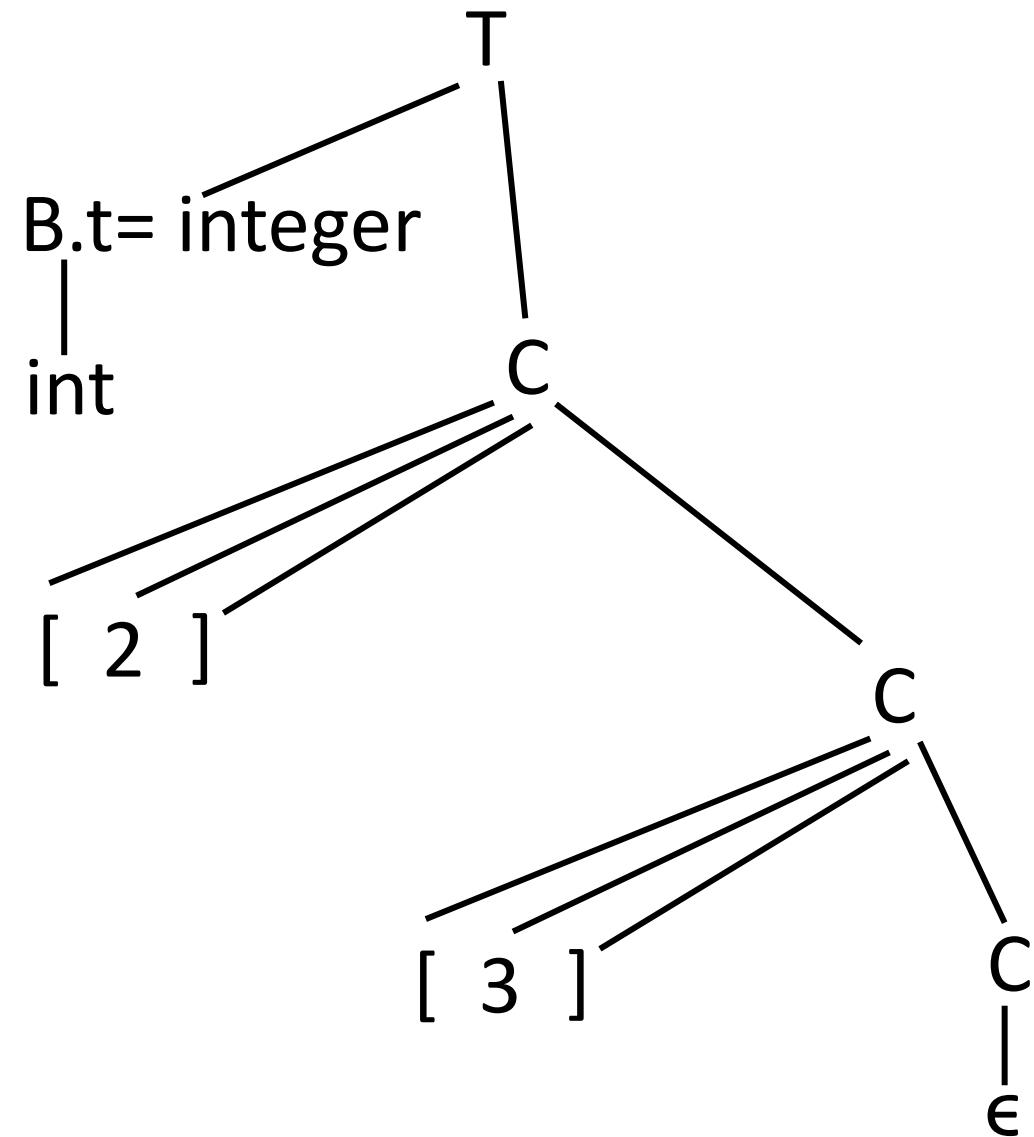
Parse Tree for  $\text{int} [2] [3]$

## Example 5.13(Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type

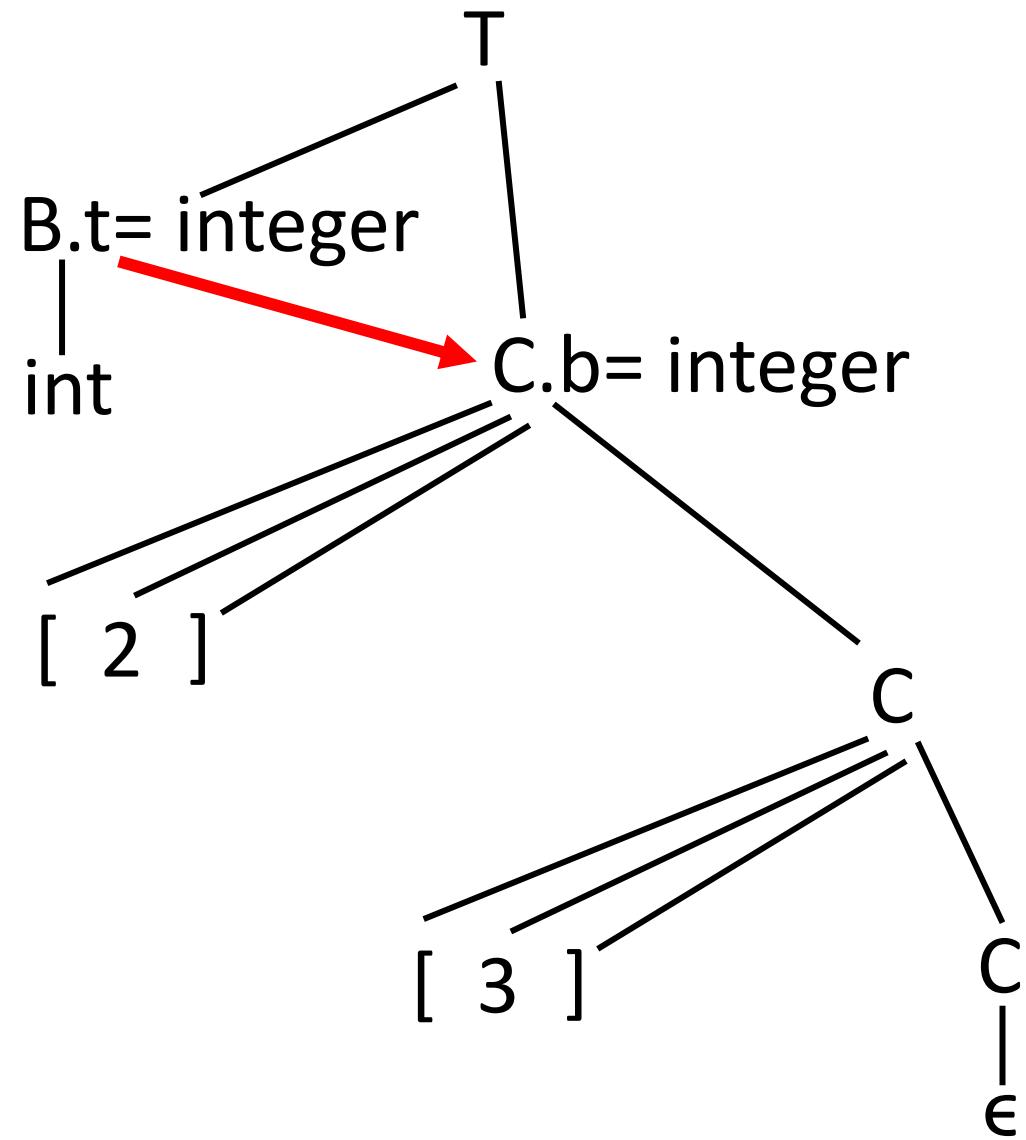


## Example 5.13 (Cont...)

### Annotated Parse Tree for int [2] [3]

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

$T$  generates either a basic type or an array type

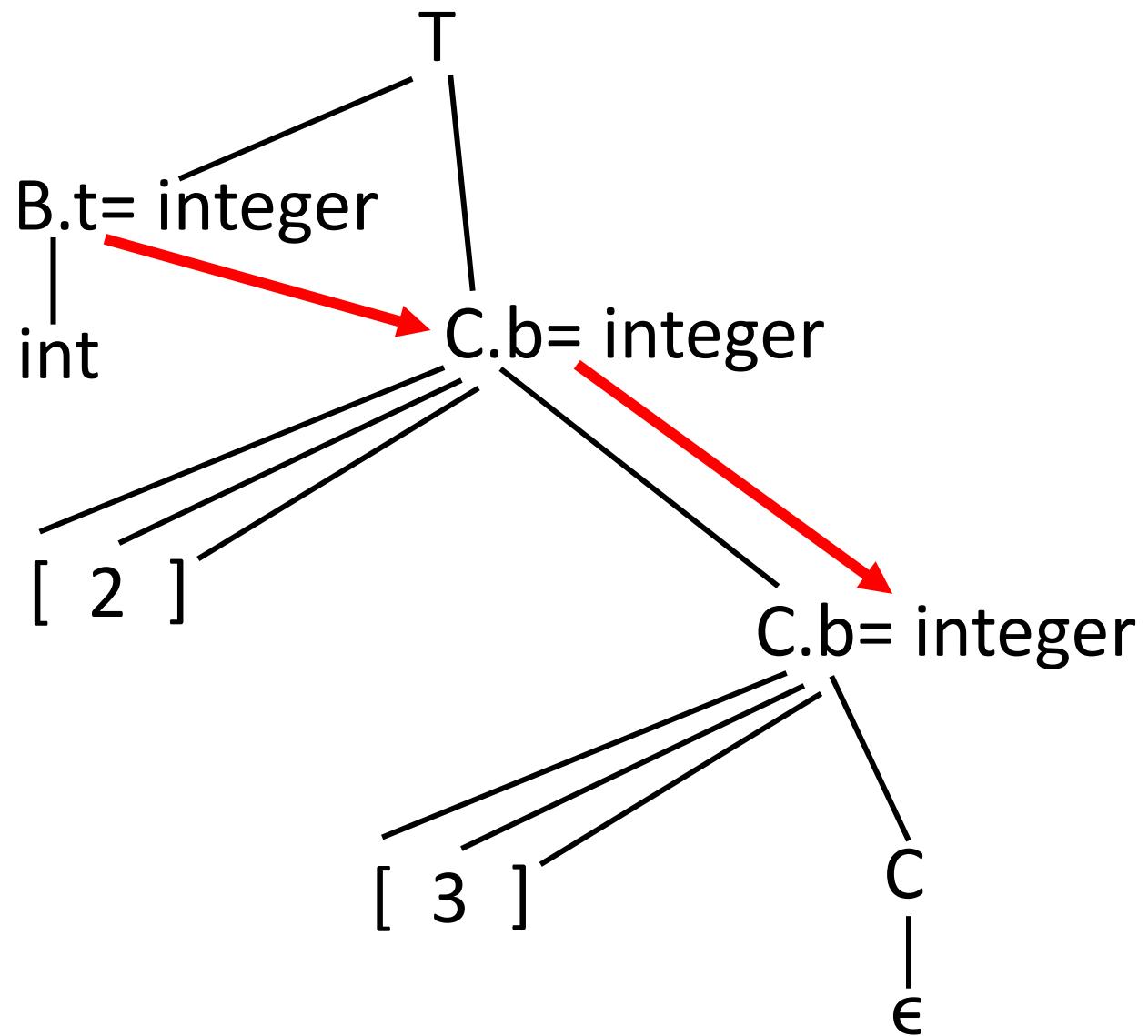


# Example 5.13 (Cont...)

## Annotated Parse Tree for int [2] [3]

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

$T$  generates either a basic type or an array type

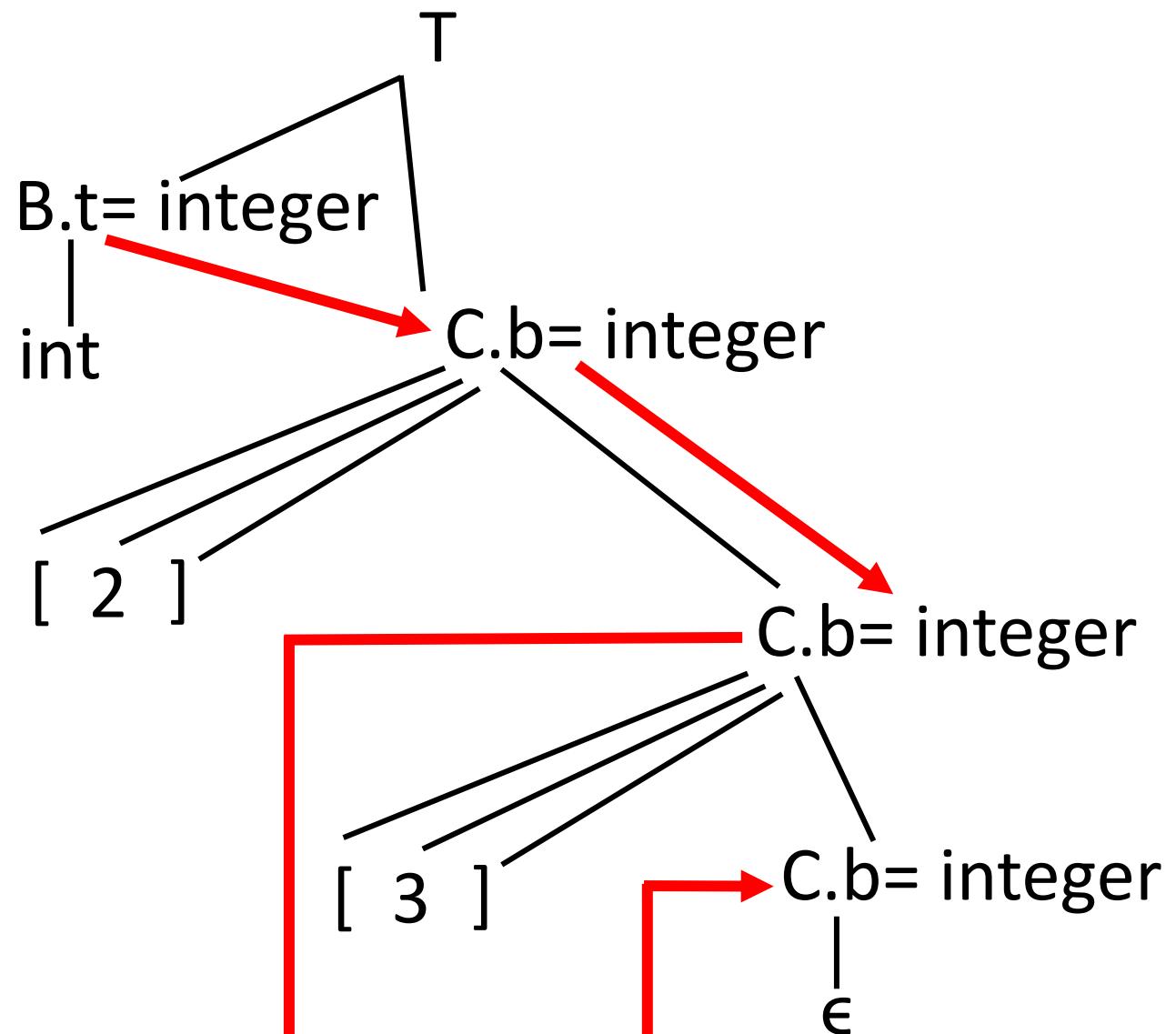


## Example 5.13 (Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type

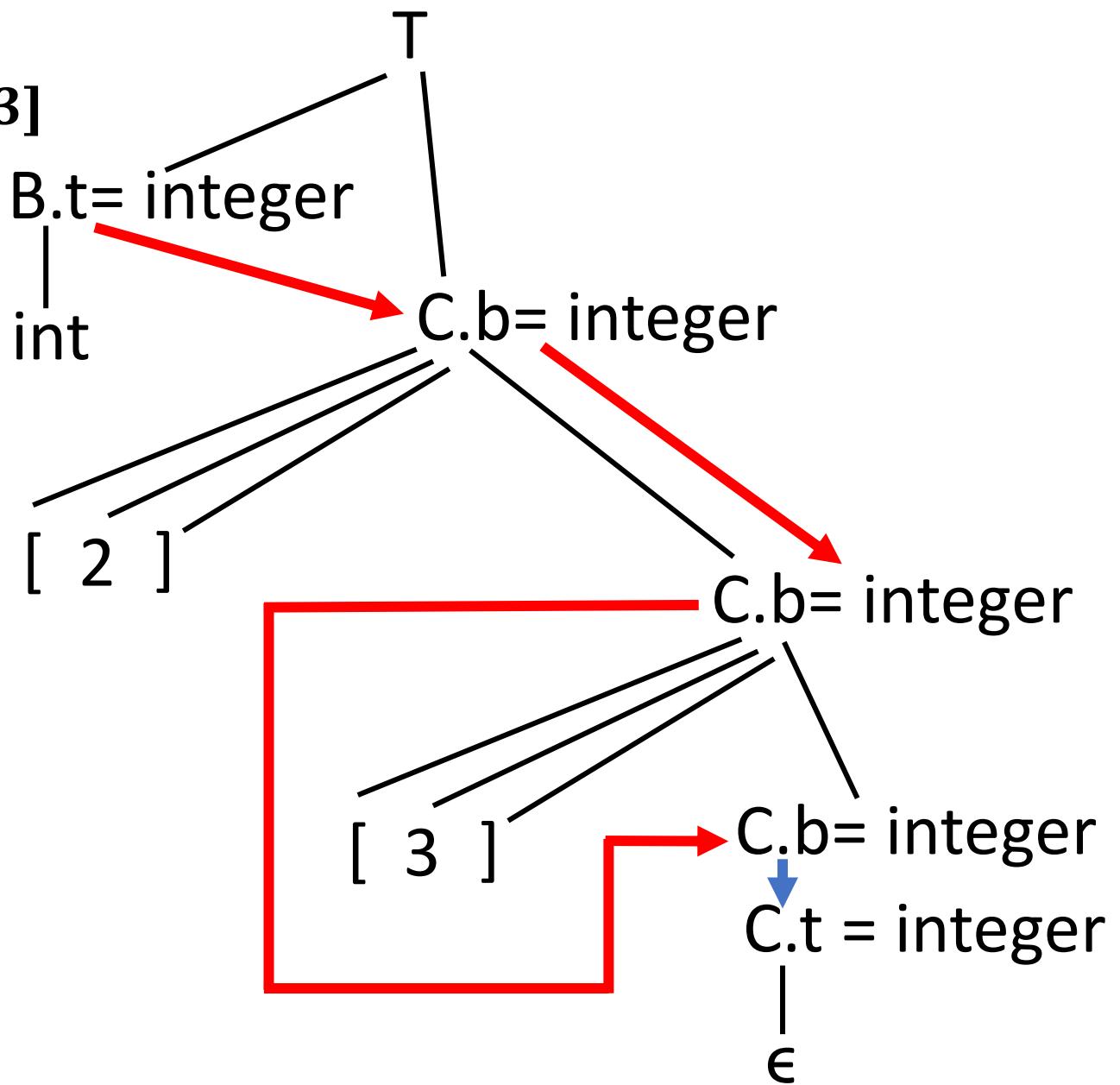


# Example 5.13 (Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type

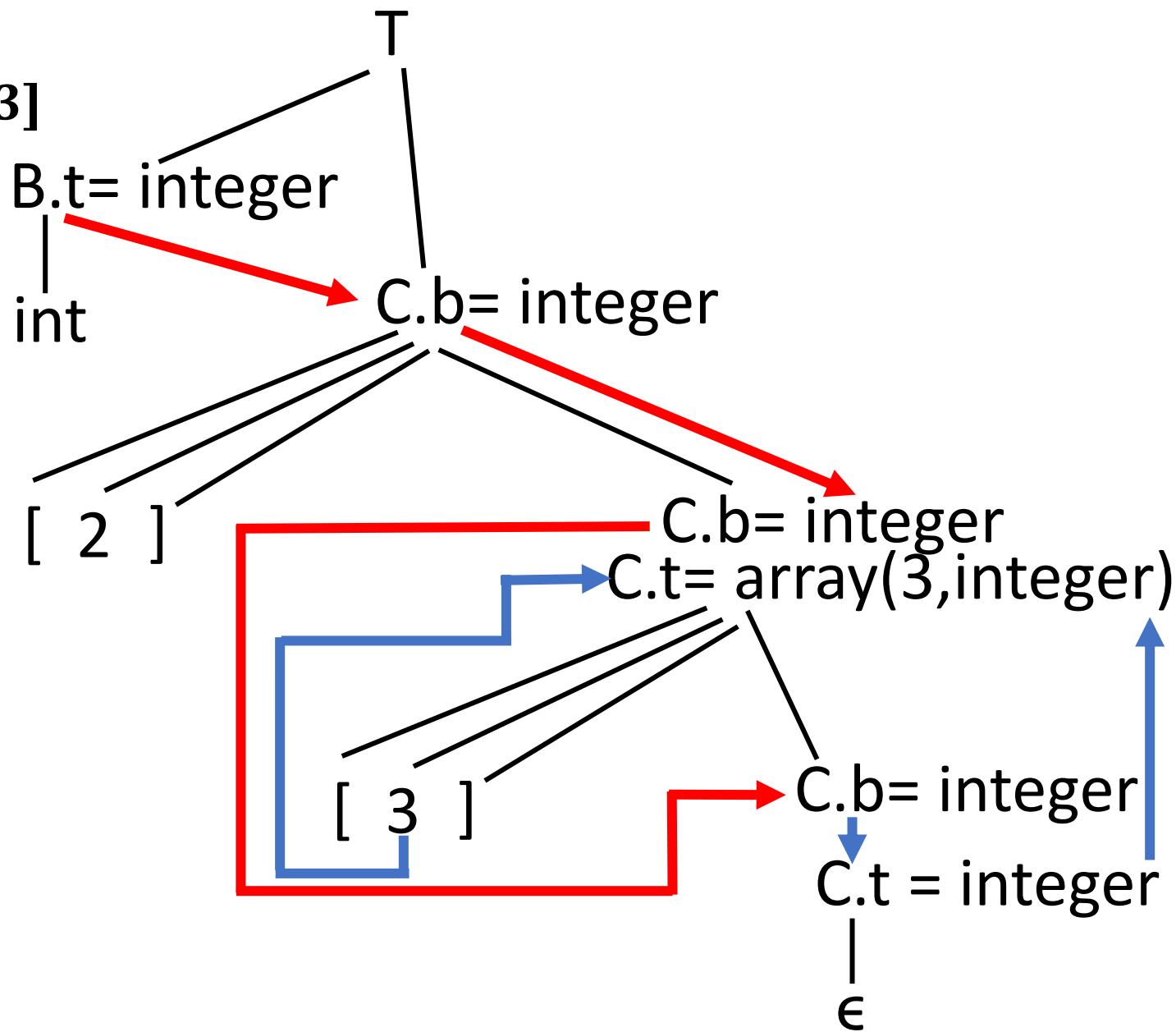


## Example 5.13 (Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type

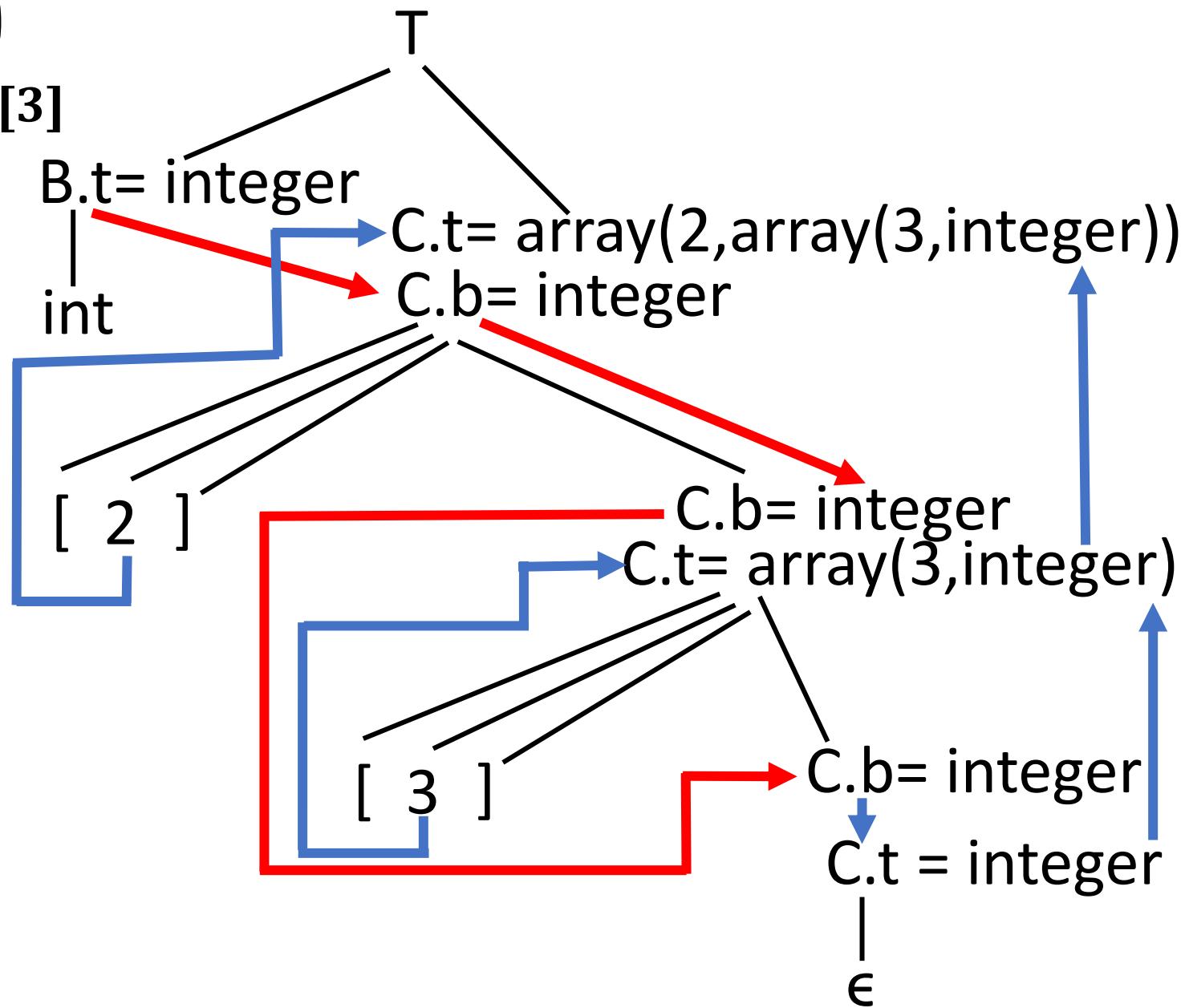


## Example 5.13 (Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type

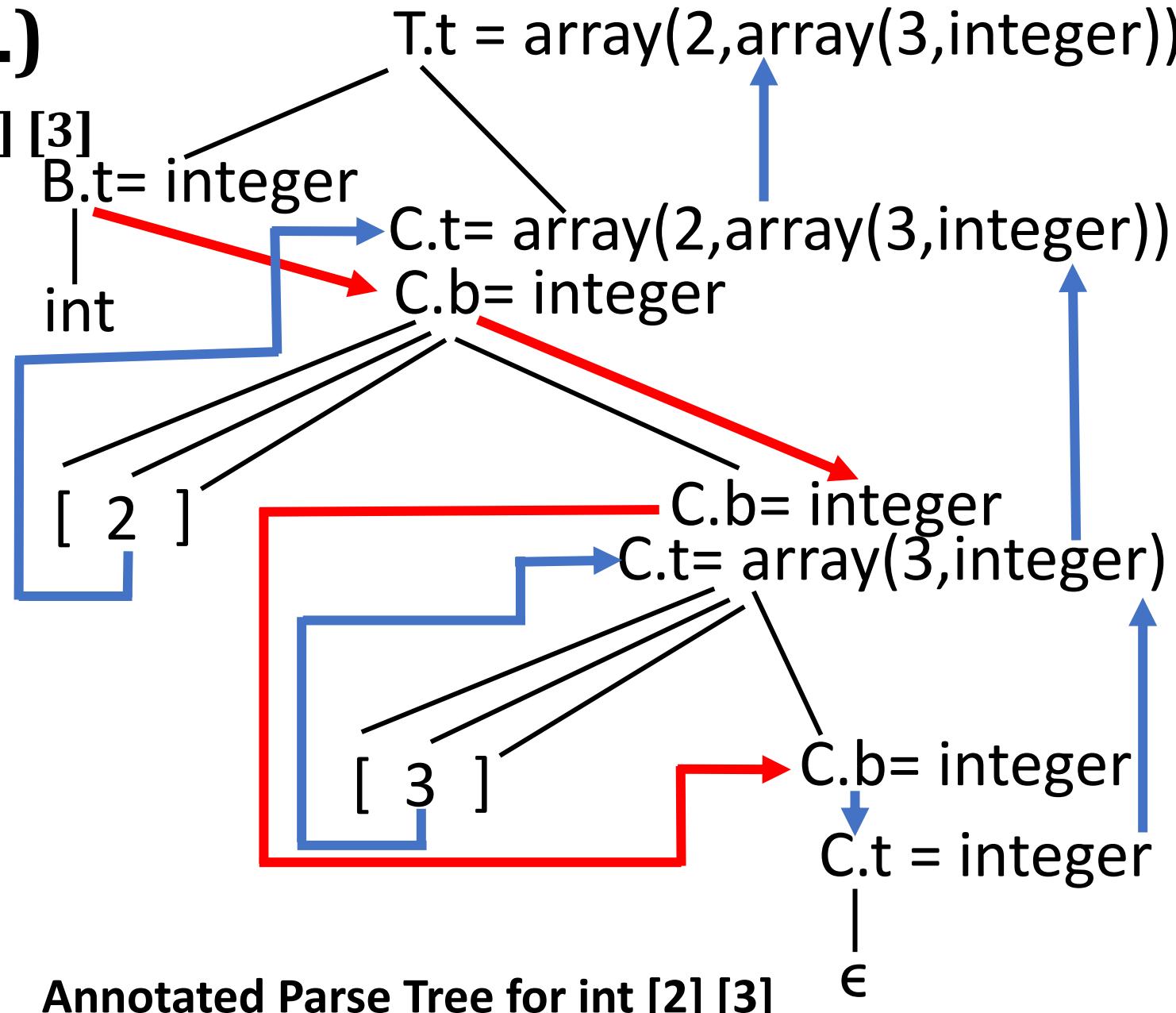


## Example 5.13 (Cont...)

Annotated Parse Tree for  $\text{int} [2] [3]$

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

$T$  generates either a basic type or an array type



# Practice Problem

- Draw Annotated Parse Tree for **3 \* 4 \* 5** by using the SDD of Example 5.3
- Draw Annotated Parse Tree for **int a,b,c,d** by using the SDD of Example 5.10
- Draw Syntax Tree for **8 + a - c** by using the SDD of Example 5.12
- Modify the SDD of Example 5.11 to construct syntax trees for a simple expression grammar involving the binary operators '\*' and '/'

# Syntax-Directed Translation Schemes

# Syntax-directed Translation Scheme

- A syntax-directed definition species the values of attributes by associating semantic rules with the grammar productions.

PRODUCTION

$$E \rightarrow E_1 + T$$

SEMANTIC RULE

$$|E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel '+'$$

- Syntax-directed definitions (SDD) can be more readable, hence more useful for **specifications**.
- However, translation schemes (SDT) can be more efficient, and hence more useful for **implementations**.

# Syntax-directed Translation Scheme

- A 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.

$$E \rightarrow E_1 + T \{ \text{print } '+' \}$$

- By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

# 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).

# SDT's With Actions Inside Productions

- 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).

# Postfix Syntax-directed Translation Scheme

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

Here, Postfix means all the semantic actions are at the last of the production body.

Fig: Postfix SDT implementing the desk calculator

# Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during [LR](#) parsing by executing the actions when reductions occur.

PRODUCTION	ACTIONS
$L \rightarrow E \text{ n}$	{ print( $stack[top - 1].val$ ); $top = top - 1;$ }
$E \rightarrow E_1 + T$	{ $stack[top - 2].val = stack[top - 2].val + stack[top].val$ ; $top = top - 2;$ }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ $stack[top - 2].val = stack[top - 2].val \times stack[top].val$ ; $top = top - 2;$ }
$T \rightarrow F$	
$F \rightarrow ( E )$	{ $stack[top - 2].val = stack[top - 1].val$ ; $top = top - 2;$ }
$F \rightarrow \text{digit}$	

## Example: 5.16

An problematic SDT that prints the prefix form of an expression, rather than evaluating the expression.

- 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)  $F \rightarrow \text{digit } \{ \text{print}(\text{digit}.lexval); \}$

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

## Example: 5.16

Fig. 5.22 shows the parse tree for expression  $3*5+4$  with actions inserted. If we visit the nodes in preorder, we get the pre-fix form of the expression:  $+ * 3 5 4$

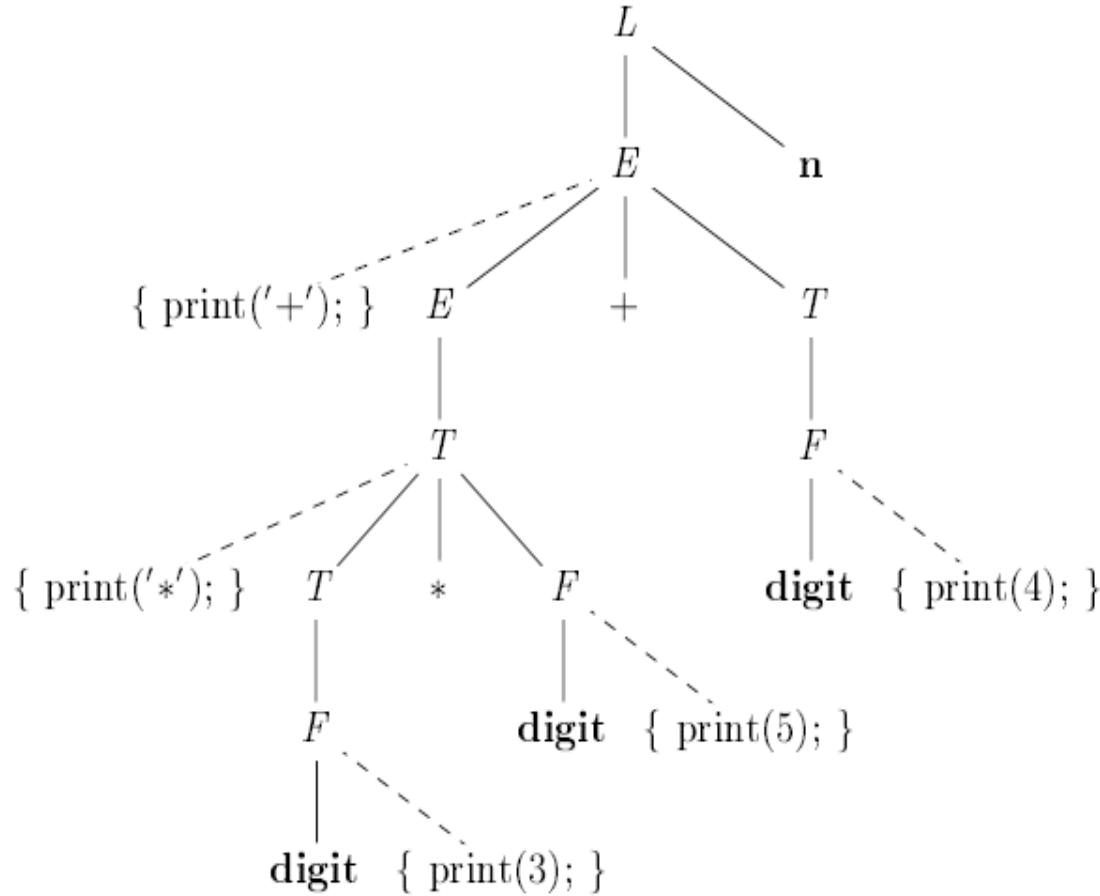


Figure 5.22: Parse tree with actions embedded

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

# Summary

- Inherited and Synthesized Attributes
- Dependency Graphs
- S-Attributed Definitions
- L-Attributed Definition
- Syntax Trees
- Syntax-Directed Translation SDT's

**End**

# Chapter-06

## Intermediate-Code Generation

# Outline

## Variants of Syntax Trees

- Directed Acyclic Graphs for Expressions
- The Value-Number Method for Constructing DAG's

## Three-Address Code

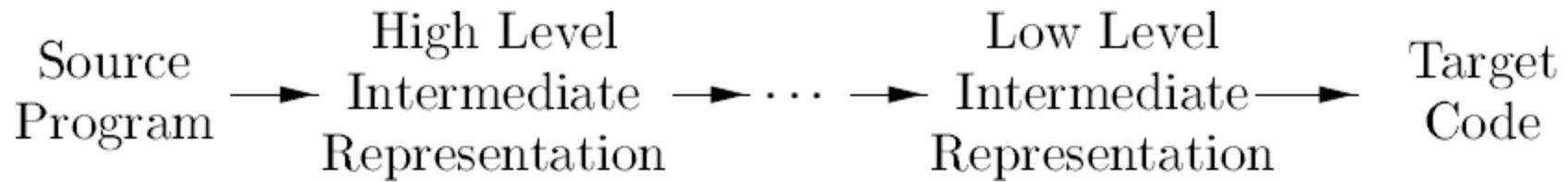
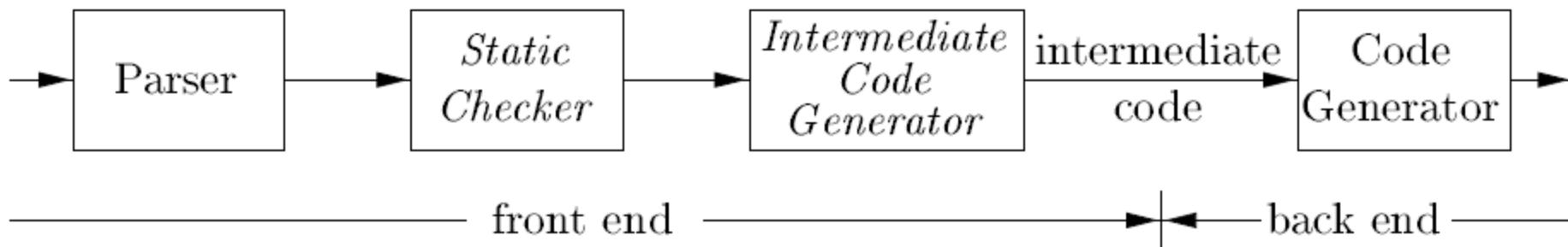
- Addresses and Instructions
- Quadruples
- Triples
- Static Single-Assignment Form

## Types and Declarations

- Type Expressions
- Type Equivalence
- Declarations
- Storage Layout for Local Names
- Sequences of Declarations

## Translation of Expressions

# Introduction



# Variants of Syntax Trees

# Variants of Syntax Trees

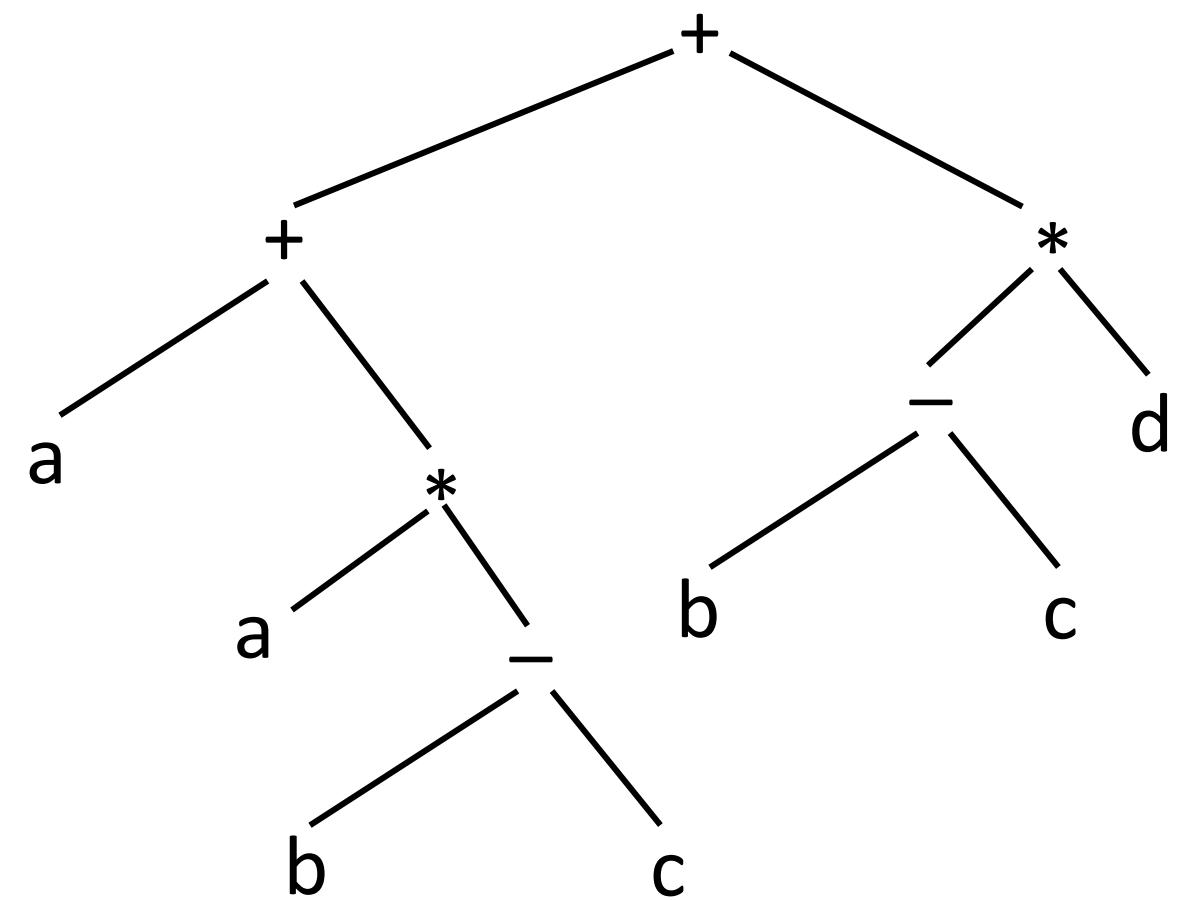
- Nodes in a syntax tree represent constructs in the source program
- The children of a node represent the meaningful components of a construct
- A **directed acyclic graph (hereafter called a DAG)** for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression
- As we shall see, DAG's can be constructed by using the same techniques that construct syntax trees

# Directed Acyclic Graphs for Expressions

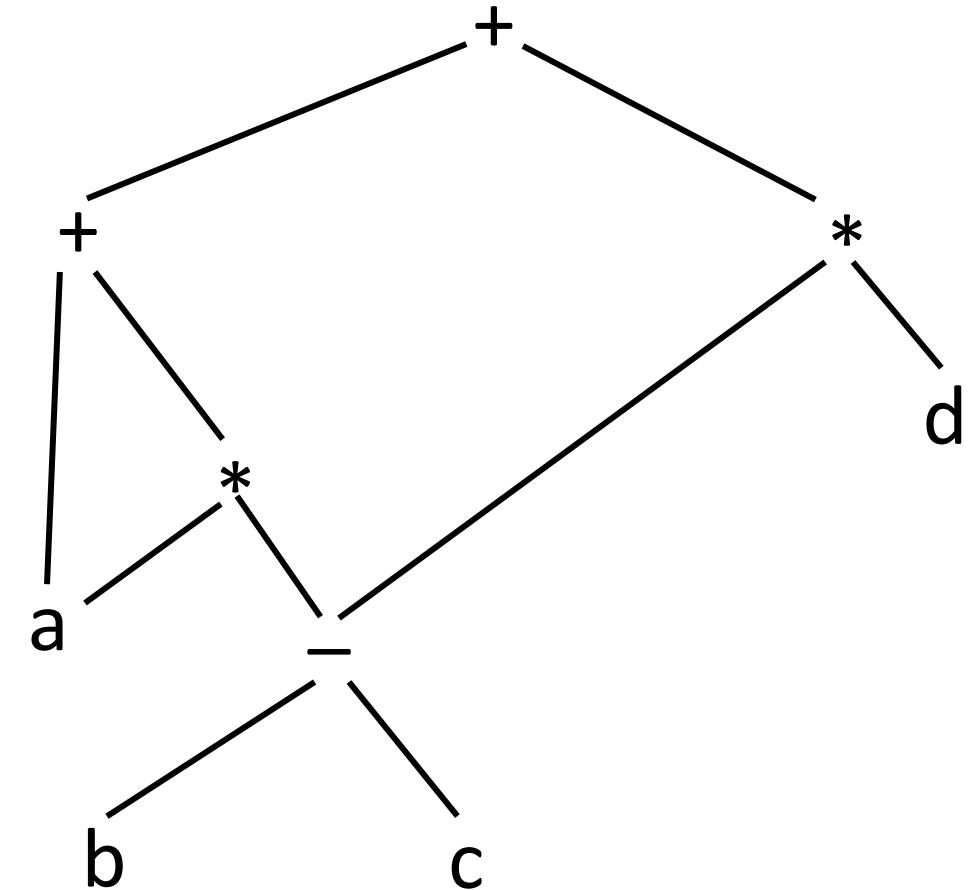
- Like the syntax tree for an expression, a DAG has **leaves** corresponding to **atomic operands** and **interior** nodes corresponding to **operators**
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression
- In a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions

## Example 6.1

Figure shows the **Syntax Tree** and **DAG** for the expression,  $a + a^*(b-c) + (b-c)^*d$



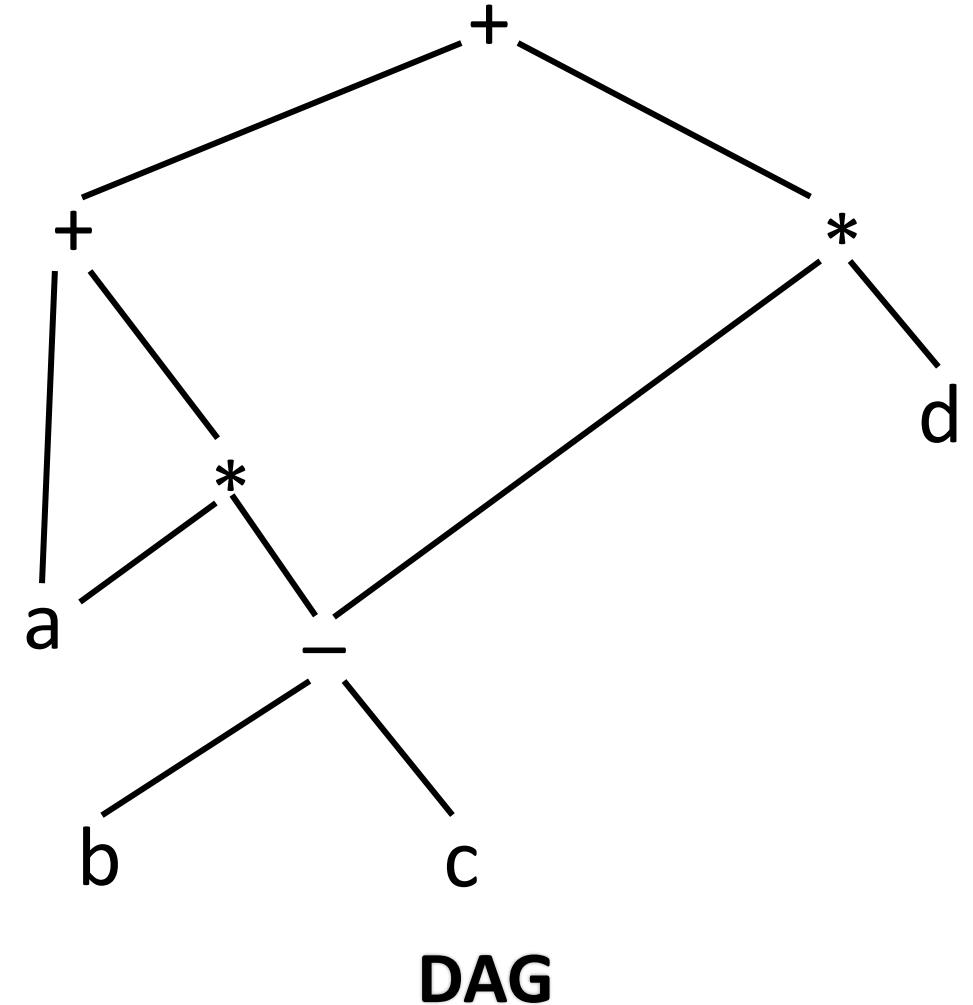
Syntax Tree



DAG

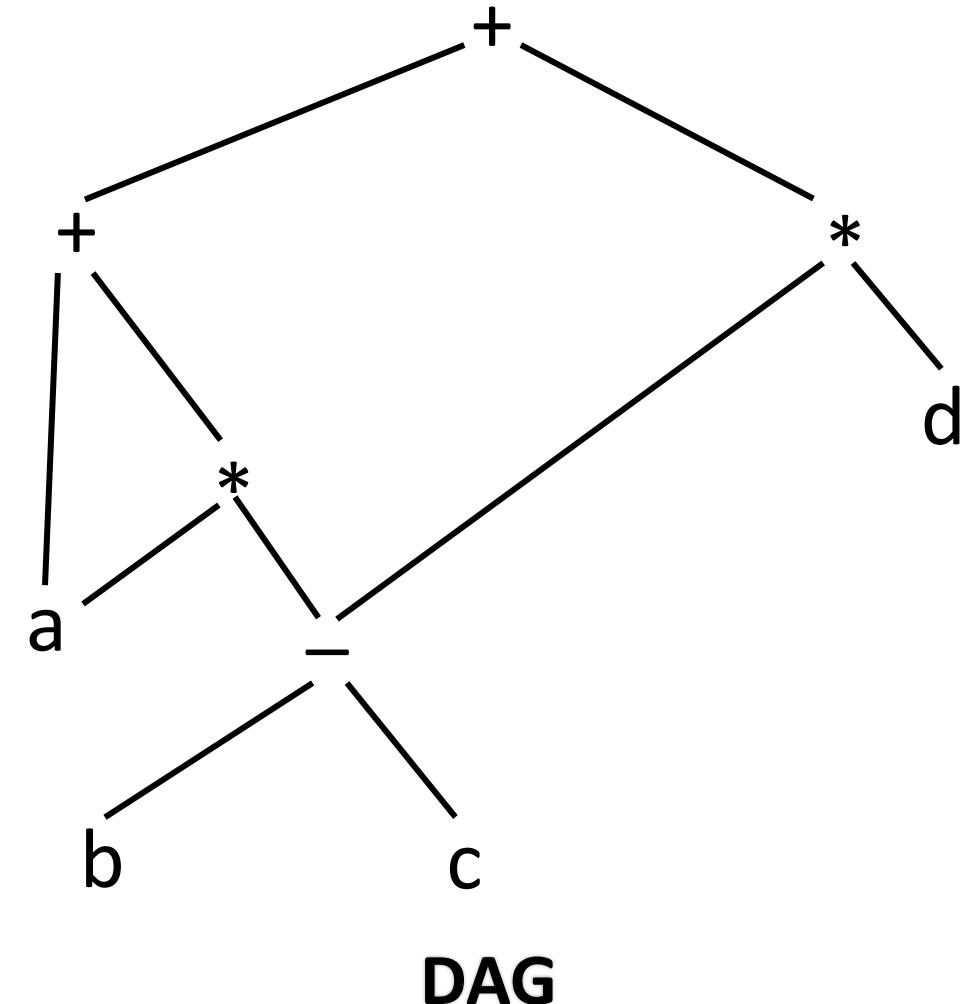
## Example 6.1 (Cont...)

- The leaf for ‘a’ has two parents, because ‘a’ appears twice in the expression
- More interestingly, the two occurrences of the common subexpression ‘b-c’ are represented by one node, the node labeled ‘\_’



## Example 6.1 (Cont...)

- That node has two parents, representing its two uses in the subexpressions ' $a*(b-c)$ ' and ' $(b-c)*d$ '
- Even though 'b' and 'c' appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression ' $b-c$ '



## Example 6.1(Cont...)

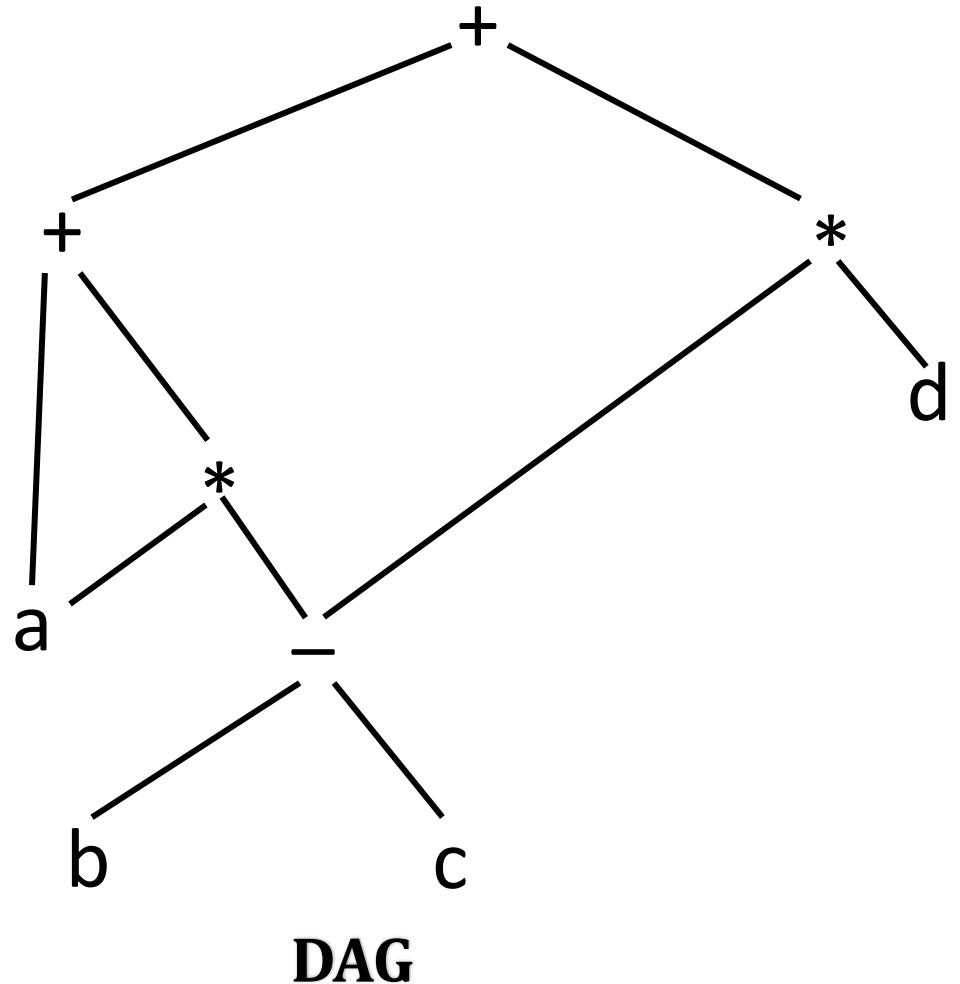
The SDD of figure can construct either syntax trees or DAG's

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.\text{node} = \mathbf{new\ Node}( '+', E_1.\text{node}, T.\text{node})$
2) $E \rightarrow E_1 - T$	$E.\text{node} = \mathbf{new\ Node}( ' - ', E_1.\text{node}, T.\text{node})$
3) $E \rightarrow T$	$E.\text{node} = T.\text{node}$
4) $T \rightarrow T_1 * F$	$T.\text{node} = \mathbf{new\ Node}( ' * ', T_1.\text{node}, F.\text{node})$
5) $T \rightarrow T_1 / F$	$T.\text{node} = \mathbf{new\ Node}( ' / ', T_1.\text{node}, F.\text{node})$
6) $T \rightarrow F$	$T.\text{node} = F.\text{node}$
7) $F \rightarrow (E)$	$F.\text{node} = E.\text{node}$
8) $F \rightarrow \mathbf{id}$	$F.\text{node} = \mathbf{new\ Leaf}( \mathbf{id}, \mathbf{id}.entry )$
9) $F \rightarrow \mathbf{num}$	$F.\text{node} = \mathbf{new\ Leaf}( \mathbf{num}, \mathbf{num}.val )$

## Example 6.2

The sequence of steps shown in figure constructs the DAG

$$a + a^*(b-c) + (b-c)^*d$$

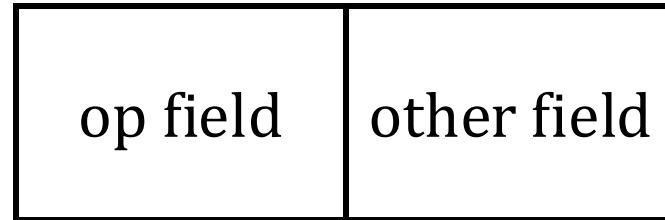


- 1)  $p_1 = \text{Leaf } (\text{id}, \text{entry-}a)$
- 2)  $p_2 = \text{Leaf } (\text{id}, \text{entry-}a) = p_1$
- 3)  $p_3 = \text{Leaf } (\text{id}, \text{entry-}b)$
- 4)  $p_4 = \text{Leaf } (\text{id}, \text{entry-}c)$
- 5)  $p_5 = \text{Node } ('-', p_3, p_4)$
- 6)  $p_6 = \text{Node } ('\ast', p_1, p_5)$
- 7)  $p_7 = \text{Node } ('+', p_1, p_6)$
- 8)  $p_8 = \text{Leaf } (\text{id}, \text{entry-}b) = p_3$
- 9)  $p_9 = \text{Leaf } (\text{id}, \text{entry-}c) = p_4$
- 10)  $p_{10} = \text{Node } ('-', p_3, p_4) = p_5$
- 11)  $p_{11} = \text{Leaf } (\text{id}, \text{entry-}d)$
- 12)  $p_{12} = \text{Node } ('\ast', p_5, p_{11})$
- 13)  $p_{13} = \text{Node } ('+', p_7, p_{12})$

Steps for Constructing DAG

# The Value-Number Method for Constructing DAG's

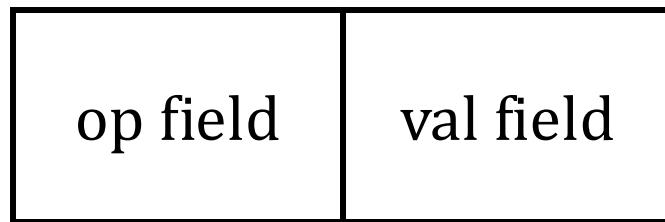
- Often, the nodes of a syntax tree or DAG are stored in an array of records
- Each row of the array represents one record, and therefore one node
- In each record, the first field is an operation code, indicating the label of the node



**Syntax Tree or DAG Node Representation as a Record**

# The Value-Number Method for Constructing DAG's (Cont...)

- Leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case)



Syntax Tree or DAG Node for a Leaf Node

# The Value-Number Method for Constructing DAG's (Cont...)

- Interior nodes have two additional fields indicating the left and right children

op field	left child	right child
----------	------------	-------------

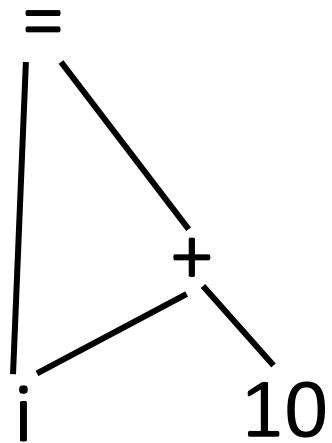
**Syntax Tree or DAG Node for an Interior Node with two children**

# The Value-Number Method for Constructing DAG's (Cont...)

- In the array, we refer to nodes by giving the integer index of the record for that node within the array
- This integer historically has been called the value number for the node or for the expression represented by the node

# The Value-Number Method for Constructing DAG's (Cont...)

DAG for  $i = i + 10$



(a) DAG

1	id			→ to entry for i
2	num	10		
3	+	1	2	
4	=	1	3	
5	...			

(b) Array

Nodes of a DAG for  $i = i + 10$  allocated in an array

## Algorithm 6.3

The value-number method for constructing the nodes of a DAG.

**INPUT:** Label  $op$ , node  $\ell$ , and node  $r$ .

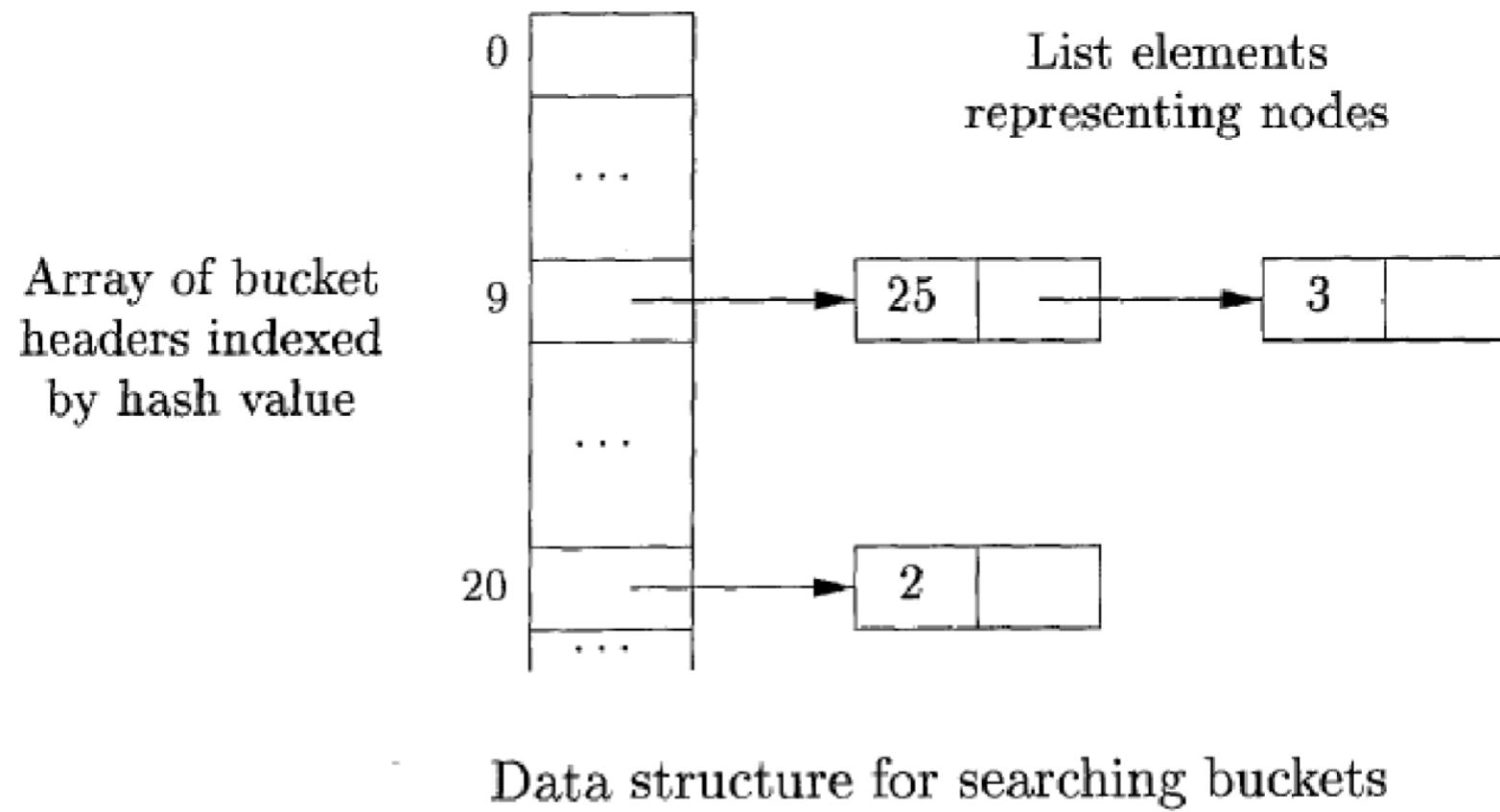
**OUTPUT:** The value number of a node in the array with signature  $\langle op, \ell, r \rangle$ .

**METHOD:** Search the array for a node  $M$  with label  $op$ , left child  $\ell$ , and right child  $r$ .

If there is such a node, return the value number of  $M$ .

If not, create in the array a new node  $N$  with label  $op$ , left child  $\ell$ , and right child  $r$ , and return its value number.

# The Value-Number Method for Constructing DAG's (Cont...)



# Three-Address Code

# Three-Address Code

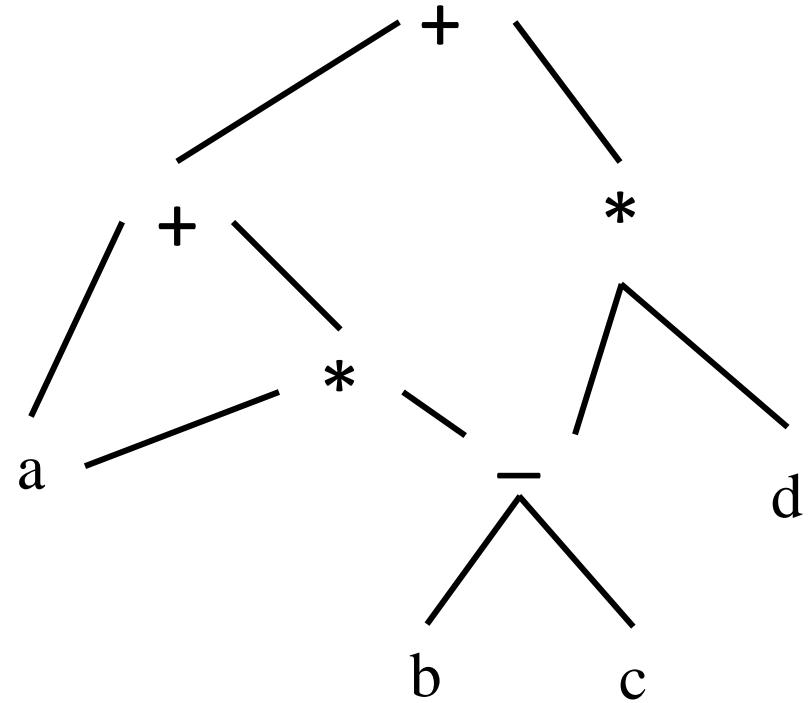
- In three-address code, there is at most one operator on the right side of an instruction
- Thus a source-language expression like ' $x + y * z$ ' might be translated into the sequence of three-address instructions
  - $t1 = y * z$
  - $t2 = x + t1$

where  $t1$  and  $t2$  are compiler-generated temporary names
- Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph

## Example 6.4

Figure shows the **DAG** and **Three-Address Code** for the expression,

$$a + a^*(b-c) + (b-c)^*d$$



(a) DAG

$$\begin{aligned}t_1 &= b - c \\t_2 &= a * t_1 \\t_3 &= a + t_2 \\t_4 &= t_1 * d \\t_5 &= t_3 + t_4\end{aligned}$$

(b) Three-Address Code

A DAG and its Corresponding Three-Address Code

# Addresses and Instructions

- Three-address code is built from two concepts: **addresses** and **instructions**
- An address can be one of the following:
  - Name
    - $a = b + c$  where **a**, **b** and **c** are source program names used as addresses
  - Constant
    - $a = 10 + 20$  where **10** and **20** are constants used as addresses
  - Compiler-generated temporary
    - $t_1 = t_2 + t_3$  where  **$t_1$** ,  **$t_2$**  and  **$t_3$**  are compiler-generated temporary used as addresses

# Addresses and Instructions (Cont...)

A list of the common three-address instruction forms:

- Assignment instructions of the form  $x = y \text{ op } z$ , where op is a binary arithmetic or logical operation, and x, y, and z are addresses  
For example: **a = b + c**
- Assignments of the form  $x = \text{op } y$ , where op is a unary operation including unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number  
For example: **a = - c ; a = (float) c**

# Addresses and Instructions (Cont...)

- Copy instructions of the form `x = y`, where x is assigned the value of y  
For example, `a = b`
- An unconditional jump `goto L` where the three-address instruction with label L is the next to be executed
- Conditional jumps of the form `if x goto L` and `ifFalse x goto L`
  - These instructions execute the instruction with label L next if x is true and false, respectively
  - Otherwise, the following three-address instruction in sequence is executed next, as usual

# Addresses and Instructions (Cont...)

- Conditional jumps such as `if x relop y goto L`, which apply a relational operator (`<`, `==`, `>=`, etc.) to `x` and `y`, and execute the instruction with label `L` next if `x` stands in relation `relop` to `y`. If not, the three-address instruction following `if x relop y goto L` is executed next, in sequence
- Procedure calls and returns are implemented using the following instructions:
  - `param x` for parameters
  - `call p,n` for procedure calls
  - `y = call p,n` for function calls
  - `return y`, where `y` representing a returned value is optional

# Addresses and Instructions (Cont...)

Example: call of the procedure

$p(x_1, x_2, \dots, x_n)$

The sequence of three-address instructions will be,

param  $x_1$

param  $x_2$

...

param  $x_n$

call  $p, n$

The integer  $n$ , indicating the number of actual parameters in  $\text{call } p, n$ , is not redundant because calls can be nested. That is, some of the first *param* statements could be parameters of a call that comes after  $p$  returns its value. That value becomes another parameter of the later call.

# Example 1 of Procedure Calls:

- Suppose that **a** is an array of integers
- **f** is a function from integers to integers
- Then, the **n = f(a[i])** assignment translate into the following three-address code:

- 1) **t<sub>1</sub> = i \* 4**
- 2) **t<sub>2</sub> = a [ t<sub>1</sub> ]**
- 3) **param t<sub>2</sub>**
- 4) **t<sub>3</sub> = call f, 1**
- 5) **n = t<sub>3</sub>**

## Example 2 of Procedure Calls:

- Now let us suppose we need to execute the nested procedure call
- Then, the **n = f(a,g(a))** assignment translate into the following three-address code:

```
param a  
t1 = call g, 1  
param t1  
t2 = call f, 2  
n = t2
```

# Addresses and Instructions (Cont...)

- Indexed copy instructions of the form  $x = y[i]$  and  $x[i] = y$ 
  - The instruction  $x = y[i]$  sets  $x$  to the value in the location  $i$  memory units beyond location  $y$
  - The instruction  $x[i] = y$  sets the contents of the location  $i$  units beyond  $x$  to the value of  $y$

# Addresses and Instructions (Cont...)

- Address and pointer assignments of the form  $x = \&y$ ,  $x = *y$  and  $*x = y$ 
  - The instruction  $x = \&y$  sets the r-value of  $x$  to be the location (l-value) of  $y$
  - Presumably  $y$  is a name and  $x$  is a pointer name
  - In the instruction  $x = *y$ , presumably  $y$  is a pointer whose r-value is a location
  - The r-value of  $x$  is made equal to the contents of that location
  - Finally  $*x = y$  sets the r-value of the object pointed to by  $x$  to the r-value of  $y$

## Example 6.5

- Consider the statement

```
do i = i+1; while (a[i] < v);
```

- Two possible translations of this statement are shown in figure

L:       $t_1 = i + 1$

$i = t_1$

$t_2 = i * 8$

$t_3 = a [ t_2 ]$

      if  $t_3 < v$  goto L

**(a) Symbolic Labels**

100:     $t_1 = i + 1$

      101:     $i = t_1$

      102:     $t_2 = i * 8$

      103:     $t_3 = a [ t_2 ]$

      104:    if  $t_3 < v$  goto 100

**(b) Position Numbers**

**Two Ways of Assigning Labels to Three-Address Statements**

# Quadruples

- The description of three-address instructions specifies the components of each type of instruction
- But it does not specify the representation of these instructions in a data structure
- In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands
- Three such representations are called “quadruples,” “triples”, and “indirect triples”

# Quadruples (Cont...)

- A quadruple (or just “quad”) has four fields, which we call `op`,  $\text{arg}_1$ ,  $\text{arg}_2$ , and `result`
- The `op` field contains an internal code for the operator
- For instance, the three-address instruction  $x = y + z$  is represented by placing `+` in `op`,  $y$  in  $\text{arg}_1$ ,  $z$  in  $\text{arg}_2$ , and  $x$  in `result`

op	$\text{arg}_1$	$\text{arg}_2$	result
<code>+</code>	$y$	$z$	$x$

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Instructions with unary operators like  $x = \text{minus } y$  or  $x = y$  do not use  $\text{arg}_2$

op	$\text{arg}_1$	$\text{arg}_2$	result
minus	y		x

# Quadruples (Cont...)

The following are some exceptions to this rule:

- For a copy statement like **x = y**, **op** is **=**, while for most other operations, the assignment operator is implied

op	arg <sub>1</sub>	arg <sub>2</sub>	result
=	y		x

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Operators like param use neither  $\text{arg}_2$  nor result. For example, [param x](#)

op	$\text{arg}_1$	$\text{arg}_2$	result
param	x		

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, [goto L](#)

op	$\text{arg}_1$	$\text{arg}_2$	result
goto			L

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, **if x goto L**

op	$\text{arg}_1$	$\text{arg}_2$	result
goto	x		L

# Quadruples (Cont...)

The following are some exceptions to this rule:

- Conditional and unconditional jumps put the target label in result
- For example, **if x < y goto L**

op	arg <sub>1</sub>	arg <sub>2</sub>	result
<	x	y	t <sub>1</sub>
goto	t <sub>1</sub>		L

## Example 6.6

Three-Address Code and Quadruples for the expression,  $a = b * -c + b * -c$

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	minus	c		$t_1$
1	*	b	$t_1$	$t_2$
2	minus	c		$t_3$
3	*	b	$t_3$	$t_4$
4	+	$t_2$	$t_4$	$t_5$
5	=	$t_5$		a
...				

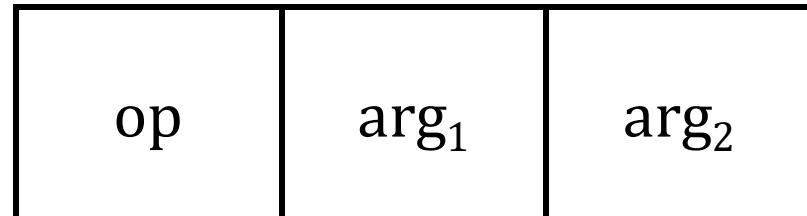
(a) Three-Address Code

(b) Quadruples

A Three-Address Code and its Corresponding Quadruples

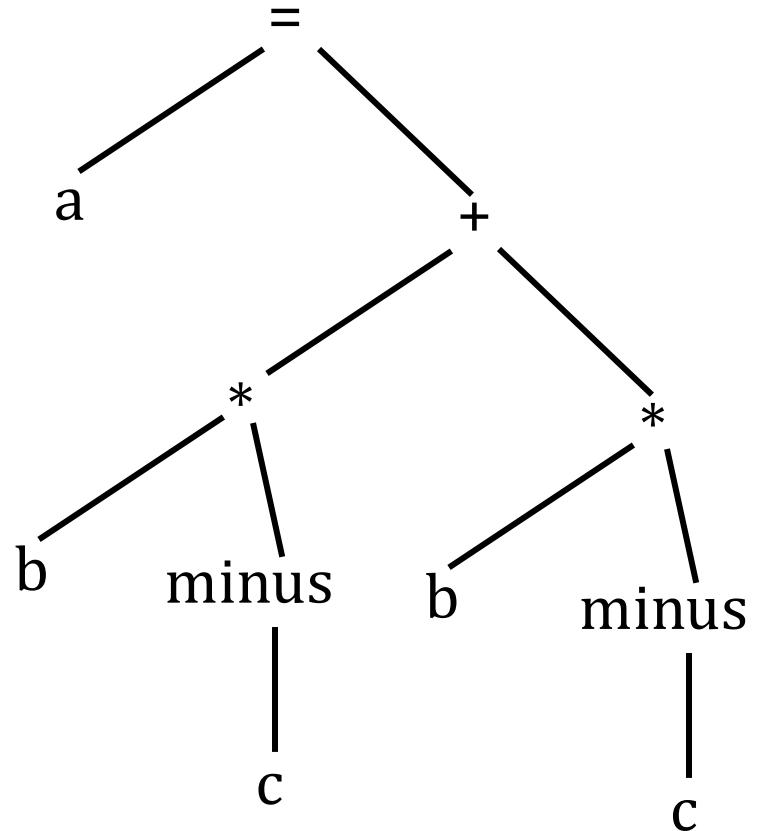
# Triples

- A triple has only **three fields**, which we call **op, arg<sub>1</sub>, arg<sub>2</sub>**
- Using triples, we refer to the result of an operation  $x \text{ op } y$  by its **position**, rather than by an explicit temporary name
- Thus, instead of the temporary  $t_1$  a triple representation would refer to position (0)
- Parenthesized numbers represent pointers into the triple structure itself



## Example 6.7

Syntax Tree and Triples for the expression, **a = b \* -c + b \* -c**



(a) Syntax Tree

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

A Syntax Tree and its Corresponding Triples

# Triples (Cont...)

- In the triple representation in (b), the copy statement `a = t5` is encoded in the triple representation by placing `a` in the `arg1` field and `(4)` in the `arg2` field

op	arg <sub>1</sub>	arg <sub>2</sub>
5 =	a	(4)

# Triples (Cont...)

- A ternary operation like  $x[i] = y$  requires **two entries** in the triple structure
- For example, we can put  $x$  and  $i$  in one triple and  $y$  in the next

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	i	4
1	+	x	(0)
2	=	(1)	y

# Triples (Cont...)

- Similarly,  $x = y[i]$  can be implemented by treating it as if it were the two instructions  $t = y[i]$  and  $x = t$ , where  $t$  is a compiler-generated temporary
- Note that the temporary  $t$  does not actually appear in a triple, since temporary values are referred to by their position in the triple structure

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	i	4
1	+	y	(0)
2	=	x	(1)

# Benefit of Quadruples over Triples

- A benefit of quadruples over triples can be seen in an [optimizing compiler](#), where instructions are often moved around
- With quadruples, if we move an instruction that computes a temporary  $t$ , then the instructions that use  $t$  require no change
- With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with [indirect triples](#), which we consider next.

# Benefit of Quadruples over Triples (Cont...)

Before Optimizing,

$$t_1 = \text{itof}(60)$$

$$t_2 = a * t_1$$

$$t_3 = t_2 + 40$$

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	itof	60		t <sub>1</sub>
1	*	a	t <sub>1</sub>	t <sub>2</sub>
2	+	t <sub>2</sub>	40	t <sub>3</sub>

**Quadruples**

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	itof	60	
1	*	a	(0)
2	+	(1)	40

**Triples**

# Benefit of Quadruples over Triples (Cont...)

After Optimizing,

$$t_2 = a * 60.0$$

$$t_3 = t_2 + 40$$

	op	arg <sub>1</sub>	arg <sub>2</sub>	result
0	*	a	60.0	t <sub>2</sub>
1	+	t <sub>2</sub>	40	t <sub>3</sub>

**Quadruples**

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	*	a	60.0
1	+	(0)	40

**Triples**

# Indirect Triples

- Indirect triples overcomes the limitation of triples
- It consists of a listing of pointers to triples, rather than a listing of triples themselves
- For example, let us use an array instruction to list pointers to triples in the desired order
- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves

# Indirect Triples(Cont...)

The triples in (b) might be represented as in this figure

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...			

instruction



(0)
(1)
(2)
(3)
(4)
(5)
...

	op	arg <sub>1</sub>	arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
...			

**(b) Triples**

**Indirect Triples Representation**

# Use of Indirect Triples by an Optimized Compiler

Before Optimizing,

$$t_1 = \text{itof}(60)$$

$$t_2 = a * t_1$$

$$t_3 = t_2 + 40$$

After Optimizing,

$$t_2 = a * 60.0$$

$$t_3 = t_2 + 40$$

(0)	op	arg <sub>1</sub>	arg <sub>2</sub>
(1)	itof	60	
(2)	*	a	(0)
...	+	(1)	40



(1)	op	arg <sub>1</sub>	arg <sub>2</sub>
(2)	*	a	60.0
...	+	(1)	40
...			

Indirect Triples

Indirect Triples

# Static Single-Assignment Form (SSA)

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations
- Two distinctive aspects distinguish SSA from three-address code
- The first is that all assignments in SSA are to variables with distinct names
- Hence the term **static single-assignment**

# Static Single-Assignment Form (Cont...)

- Following shows the same intermediate program in three-address code and in static single assignment form
- Note that subscripts distinguish each definition of variables p and q in the SSA representation

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$

**(a) Three-Address Code**

Intermediate Program in Three-Address Code and SSA

**(b) Static Single-Assignment Form**

# Static Single-Assignment Form (Cont...)

- The same variable may be defined in two different control-flow paths in a program
- For example, the source program

```
if ( flag ) x = -1; else x = 1;  
y = x * a;
```

has two control-flow paths in which the variable **x** gets defined

- If we use different names for **x** in the true part and the false part of the conditional statement, then which name should we use in the **assignment y = x \* a?**

# Static Single-Assignment Form (Cont...)

- Here is where the second distinctive aspect of SSA comes into play
- SSA uses a notational convention called the  $\phi$ -function to combine the two definitions of  $x$ :

```
if ( flag ) x1 = -1; else x2 = 1;  
x3 = φ(x1, x2)  
y = x3* a;
```

- Here,  $\phi(x_1, x_2)$  has the value  $x_1$  if the control flow passes through the true part of the conditional and the value  $x_2$  if the control flow passes through the false part.
- That is to say, the  $\phi$ -function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the  $\phi$ -function

# Practice Problem-1

Expression:  $((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$

- Construct for the above expression by maintaining precedence
  - Syntax Tree
  - DAG
  - Allocate the Nodes of the DAG in an Array by using Value-Number Method
  - Three-Address Code
  - Quadruples
  - Triples
  - Indirect Triples
  - SSA

# Types and Declarations

# Types and Declarations

- The applications of types can be grouped under checking and translation
  - **Type checking** uses logical rules to reason about the behavior of a program at run time
  - Specifically, it ensures that the types of the operands match the type expected by an operator
  - For example, the `&&` operator in Java expects its two operands to be booleans and the result is also of type Boolean

# Types and Declarations(Cont...)

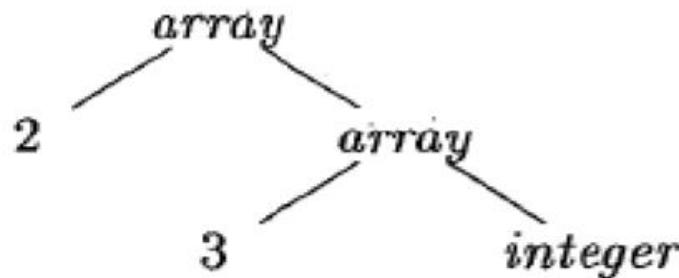
- The applications of types can be grouped under checking and translation
  - **Translation Applications:** From the type of a name, a compiler can determine the storage that will be needed for that name at run time
  - Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions etc.

# Type Expressions

- Types have structure, which we shall represent using type expressions
- A type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression
- The sets of basic types and constructors depend on the language to be checked

## Example 6.8

- 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

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - **A basic type is a type expression**
  - Typical basic types for a language include boolean, char, integer, float, and void
  - “void” denotes “the absence of a value”
  - All these are examples of basic types

float x;	//Type Expression: float
int i;	//Type Expression: int
float m;	//Type Expression: float

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - A **type name** is a type expression
  - A **type expression** can be formed by applying the array type constructor to a number and a type expression
  - All these are examples of array type constructor applied to a number and a type expression

array int[2];

//Type Expression: array(2,int)

array float[2][3];

//Type Expression: array(2,array(3,float))

array float[2][3][4];

//Type Expression: array(2,array(3,array(4,float)))

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - A record is a data structure with named fields
  - A type expression can be formed by applying the record type constructor to the field names and their types
  - The use of a name x for a field within a record does not conflict with other uses of the name outside the record
  - Thus, the three uses of x in the following declarations are distinct and do not conflict with each other:

```
float x;           //Type Expression: float
struct {float x; float y;} p;    //Type Expression: record(x:float,y:float)
struct {int t; float x; float y;} q; //Type Expression: record(t:int,x:float,y:float)
```

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - A type expression can be formed by using the type constructor  $\rightarrow$  for function types
  - We write  $s \rightarrow t$  for “function from type  $s$  to type  $t$ ”
  - Here the functions map one type expression to the other

function int f(float x); Type Expression: float  $\rightarrow$  int

function float g(int a, float b, int c);

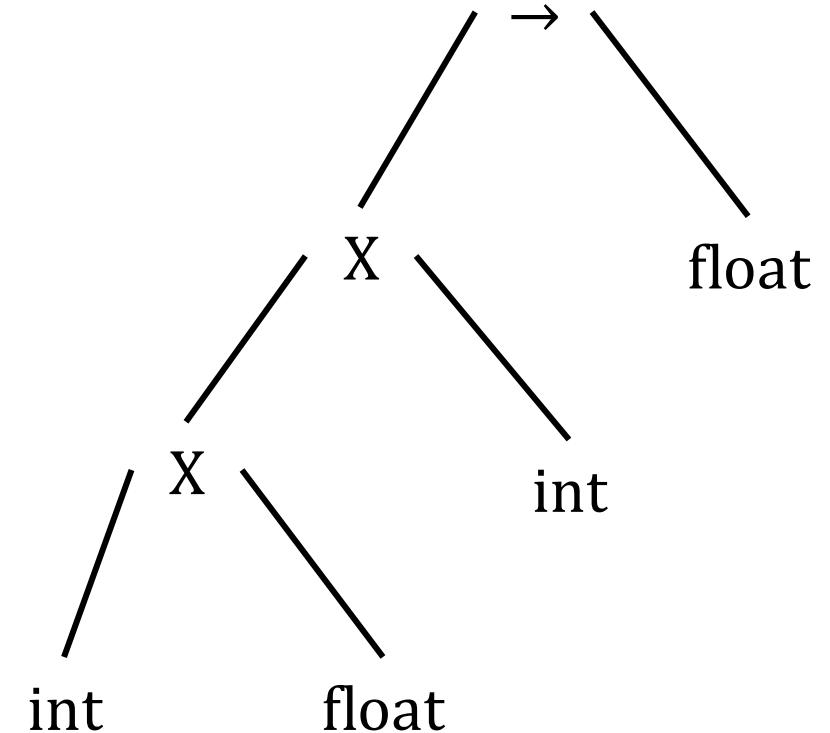
Type Expression: int  $\times$  float  $\times$  int  $\rightarrow$  float

# Type Expressions (Cont...)

- We shall use the following definition of type expressions:
  - If  $s$  and  $t$  are type expressions, then their Cartesian product  $s \times t$  is a type expression
  - They can be used to represent function parameters
  - We assume that  $\times$  associates to the left and that it has higher precedence than  $\rightarrow$
  - **Type expressions may contain variables whose values are type expressions**

# Type Expressions (Cont...)

- A convenient way to represent a type expression is to use a graph
- Example: Type expression for  
**function float g(int a, float b, int c);**  
is **int × float × int → float**
- It can be represented by generating **expression tree** like below
- **Interior Nodes:** type constructor
- **Leaves:** basic types, types names and type variables



# Type Equivalence

- When are two type expressions **equivalent**?
- Many type-checking rules have the form, “**if** two type expressions are equal **then** return a certain type **else** error.”
- Two types can be either
  - Structurally Equivalent or
  - Name Equivalent

# Type Equivalence (Cont...)

- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:
  - They are the same basic type
  - They are formed by applying the same constructor to structurally equivalent types
  - One is a type name that denotes the other
- If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions

# Name Equivalence

- **Name Equivalence:** two types are equal if and only if they have the same name
- Thus, for example in the code (using C syntax)

```
struct ST{
```

```
    int a;
```

```
    float b;
```

```
}X,Y;
```

```
struct T{
```

```
    int a;
```

```
    float b;
```

```
}M,N;
```

# Name Equivalence (Cont...)

```
struct ST{  
    int a;  
    float b;  
}X,Y;
```

```
struct T{  
    int a;  
    float b;  
}M,N;
```

- If name equivalence is used in the language then
  - **X and Y** would be of the same type and
  - **M and N** would be of the same type but
  - The type of **X or Y** would not be equivalent to the type of **M or N**
- This means that statements such as:
  - **X = Y** and **M = N** would be valid but
  - **X = M** would not be valid (would not be accepted by a translator)

# Structural Equivalence

- **Structural Equivalence:** two types are equal if and only if, they have the same structure which can be interpreted in different ways
  - A strict interpretation would be that the names and types of each component of the two types must be the same and must be listed in the same order in the type definition
  - A less stringent requirement would be that the component types must be the same and in the same order in the two types, but the names of the components could be different

# Structural Equivalence (Cont...)

```
struct ST{  
    int a;  
    float b;  
}X,Y;
```

```
struct T{  
    int a;  
    float b;  
}M,N;
```

- Again looking at the example above using structural equivalence the two types **ST** and **T** would be considered equivalent which means that a translator would accept statements such as **X = M**
- Note that C doesn't support structural equivalence for struct and classes and will give error for above assignment

# Declarations

- Following grammar consisting of types and declarations that declares just one name at a time

$$D \rightarrow T \mathbf{id} ; D \mid \epsilon$$
$$T \rightarrow B C \mid \mathbf{record} ' \{ ' D ' \}'$$
$$B \rightarrow \mathbf{int} \mid \mathbf{float}$$
$$C \rightarrow \epsilon \mid [ \mathbf{num} ] C$$

# Declarations (Cont...)

$$D \rightarrow T \mathbf{id} ; D \mid \epsilon$$
$$T \rightarrow B C \mid \mathbf{record}'\{ D '\}'$$
$$B \rightarrow \mathbf{int} \mid \mathbf{float}$$
$$C \rightarrow \epsilon \mid [ \mathbf{num} ] C$$

- Nonterminal D generates a sequence of declarations
- Nonterminal T generates basic, array or record types
- Nonterminal B generates one of the basic types int and float

# Declarations (Cont...)

$D \rightarrow T \text{ id} ; D \mid \epsilon$

$T \rightarrow BC \mid \text{record} \{ D \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \epsilon \mid [ \text{num} ] C$

- Nonterminal  $C$  for “component” generates strings of zero or more integers and each integer surrounded by brackets
- An array type consists of a basic type specified by  $B$  followed by array components specified by nonterminal  $C$
- A record type (the second production for  $T$ ) is a sequence of declarations for the fields of the record and all surrounded by curly braces

## Practice Problem-2

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$
$$T \rightarrow B C \mid \text{record} \{ ' D ' \}$$
$$B \rightarrow \text{int} \mid \text{float}$$
$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

- For the grammar given above, construct parse tree for the following input strings:
  - **int a; float b;**
  - **int [3] [4] a;**
  - **record { int a; float b; int [3] [4] a; }**

# Practice Problem-3

$$D \rightarrow T \text{ id} ; D \mid \epsilon$$
$$T \rightarrow B C \mid \text{record} \{ ' D ' \}$$
$$B \rightarrow \text{int} \mid \text{float}$$
$$C \rightarrow \epsilon \mid [ \text{num} ] C$$

- Modify the grammar given above so that it can construct parse tree for declarations with list of names and then construct parse tree for the following input
  - **record { int a, b; int [3] [4] a; }**

[Hint: Use Example 5.10]

# Storage Layout for Local Names

- From the type of a name, we can determine the amount of storage that will be needed for the name at run time
- At compile time, we can use these amounts to assign each name a relative address
- The type and relative address are saved in the symbol-table entry for the name
- Data of varying length, such as strings or data whose size cannot be determined until run time, such as dynamic arrays is handled by reserving a known fixed amount of storage for a pointer to the data

# Storage Layout for Local Names (Cont...)

- Suppose storage comes in blocks of contiguous bytes where byte is the smallest unit of addressable memory
- Typically a byte is eight bits, and some number of bytes form a machine word
- Multibyte objects are stored in consecutive bytes and given the address of the first byte
- The width of a type is the number of storage units needed for objects of that type

# Storage Layout for Local Names (Cont...)

- The syntax directed translation scheme (SDT) in figure computes types and their widths for basic and array types

- 1)  $T \rightarrow B$        $\{ t = B.type; w = B.width; \}$   
                         $C$        $\{ T.type = C.type; T.width = C.width; \}$
- 2)  $B \rightarrow \text{int}$        $\{ B.type = \text{integer}; B.width = 4; \}$
- 3)  $B \rightarrow \text{float}$        $\{ B.type = \text{float}; B.width = 8; \}$
- 4)  $C \rightarrow \epsilon$        $\{ C.type = t; C.width = w; \}$
- 5)  $C \rightarrow [\text{num}] C_1$        $\{ C.type = \text{array}(\text{num.value}, C_1.type);$   
                                 $C.width = \text{num.value} \times C_1.width; \}$

**Computing types and their width**

# Example 6.9 (Cont...)

## SDT of Array Types for int [2] [3]

$T \rightarrow B$  {  $t = B.type; w = B.width;$  }

$C$  {  $T.type = C.type; T.width = C.width;$  }

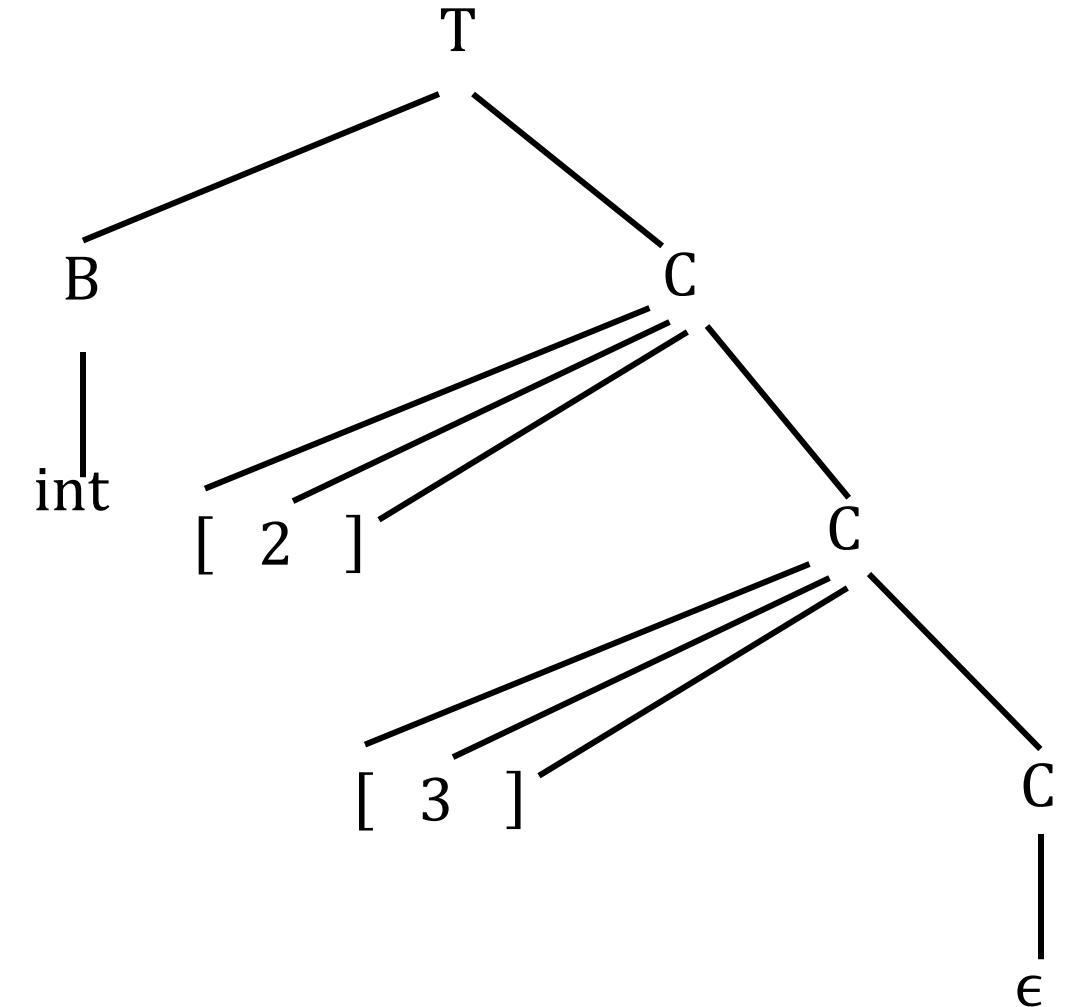
$B \rightarrow \text{int}$  {  $B.type = \text{integer}; B.width = 4;$  }

$B \rightarrow \text{float}$  {  $B.type = \text{float}; B.width = 8;$  }

$C \rightarrow \epsilon$  {  $C.type = t; C.width = w;$  }

$C \rightarrow [\text{num}] C_1$  {  $C.type = \text{array}(\text{num.value}, C_1.type);$   
 $C.width = \text{num.value} \times C_1.width;$  }

Computing types and their width

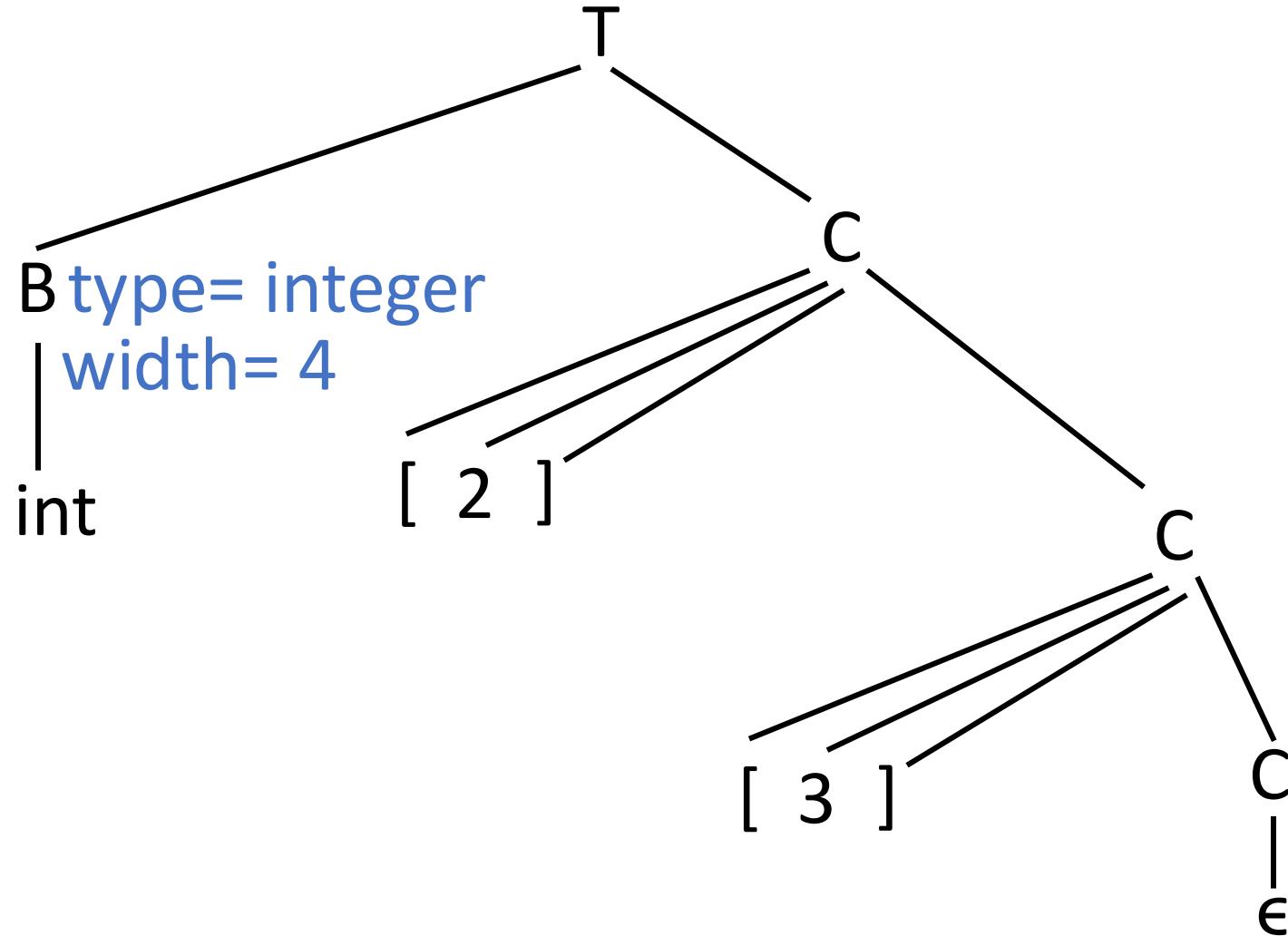


Parse Tree for int [2] [3]

## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

T → B	{ t = B.type; w = B.width; }
C	{ T.type = C.type; T.width = C.width; }
B → int	{ B.type = integer; B.width = 4; }
B → float	{ B.type = float; B.width = 8; }
C → ε	{ C.type = t; C.width = w; }
C → [num] C <sub>1</sub>	{ C.type = array(num.value, C <sub>1</sub> .type); C.width = num.value X C <sub>1</sub> .width; }



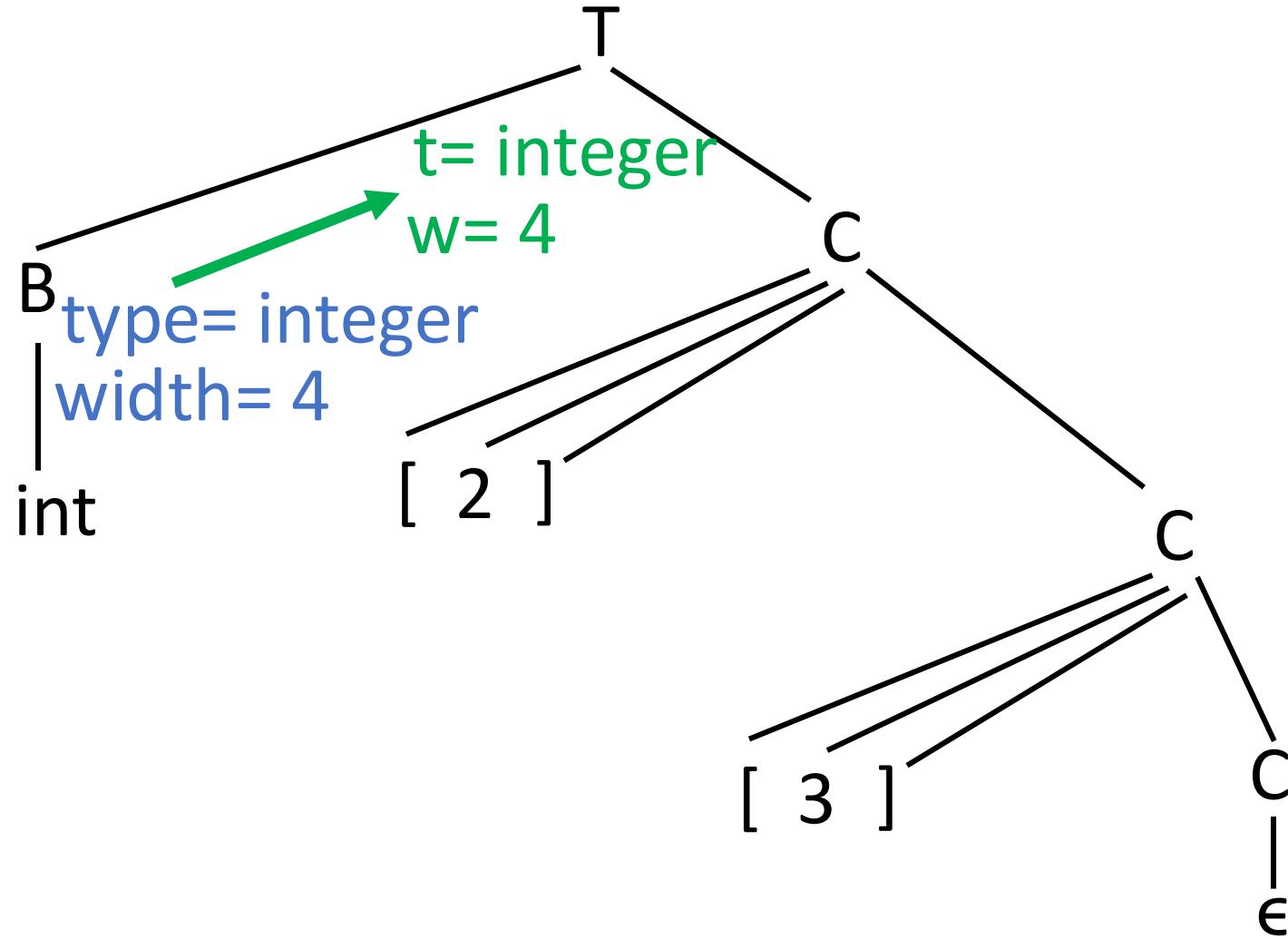
Computing types and their width

## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

T → B       $\{ t = B.type; w = B.width; \}$   
C       $\{ T.type = C.type; T.width = C.width; \}$   
B → int       $\{ B.type = integer; B.width = 4; \}$   
B → float       $\{ B.type = float; B.width = 8; \}$   
C → ε       $\{ C.type = t; C.width = w; \}$   
C → [num] C<sub>1</sub>       $\{ C.type = array(num.value, C_1.type); C.width = num.value \times C_1.width; \}$

Computing types and their width

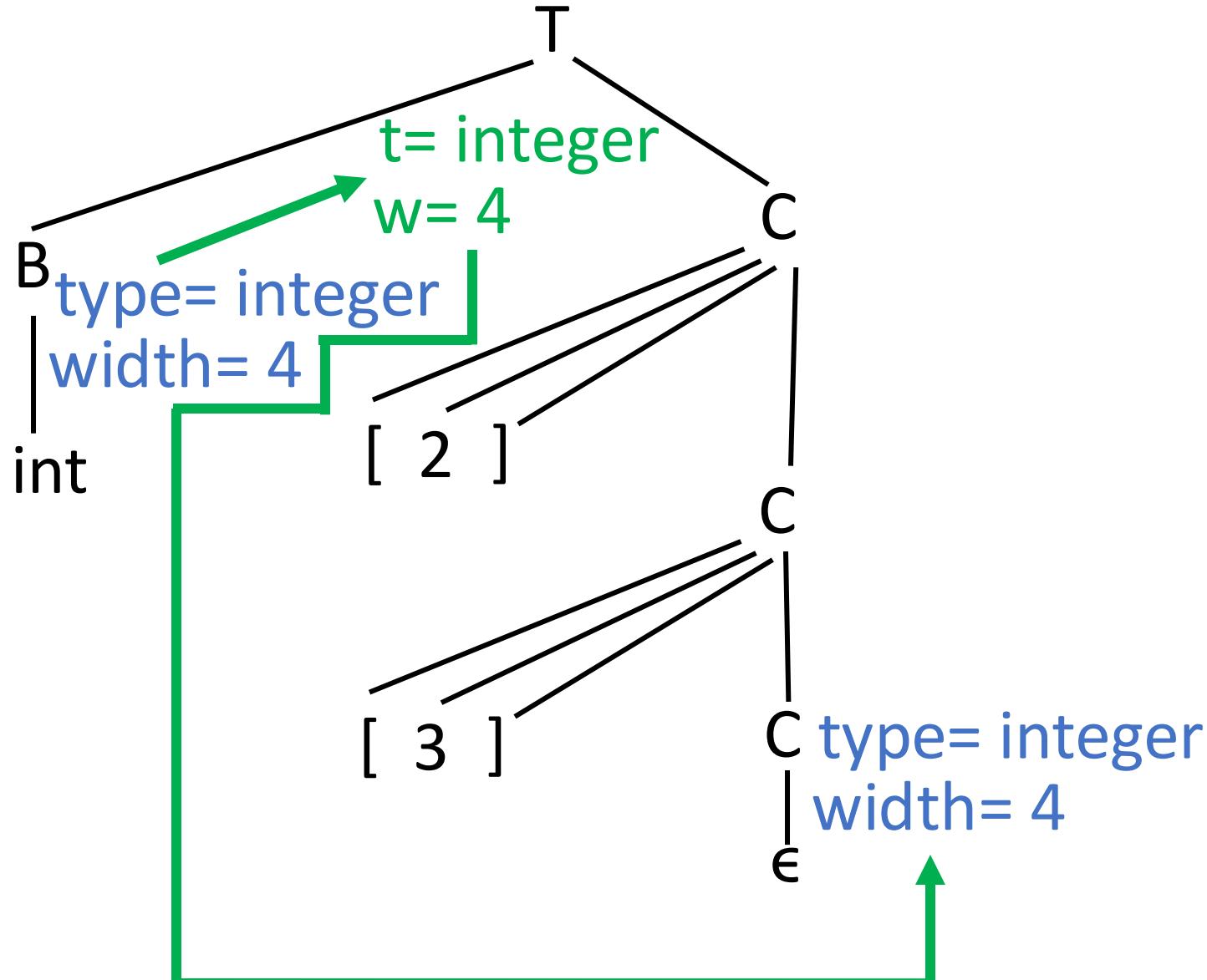


## Example 6.9 (Cont...)

SDT of Array Types for int [2] [3]

T → B       $\{ t = B.type; w = B.width; \}$   
C       $\{ T.type = C.type; T.width = C.width; \}$   
B → int       $\{ B.type = integer; B.width = 4; \}$   
B → float       $\{ B.type = float; B.width = 8; \}$   
C → ε       $\{ C.type = t; C.width = w; \}$   
C → [num] C<sub>1</sub>       $\{ C.type = array(num.value, C_1.type); C.width = num.value \times C_1.width; \}$

Computing types and their width

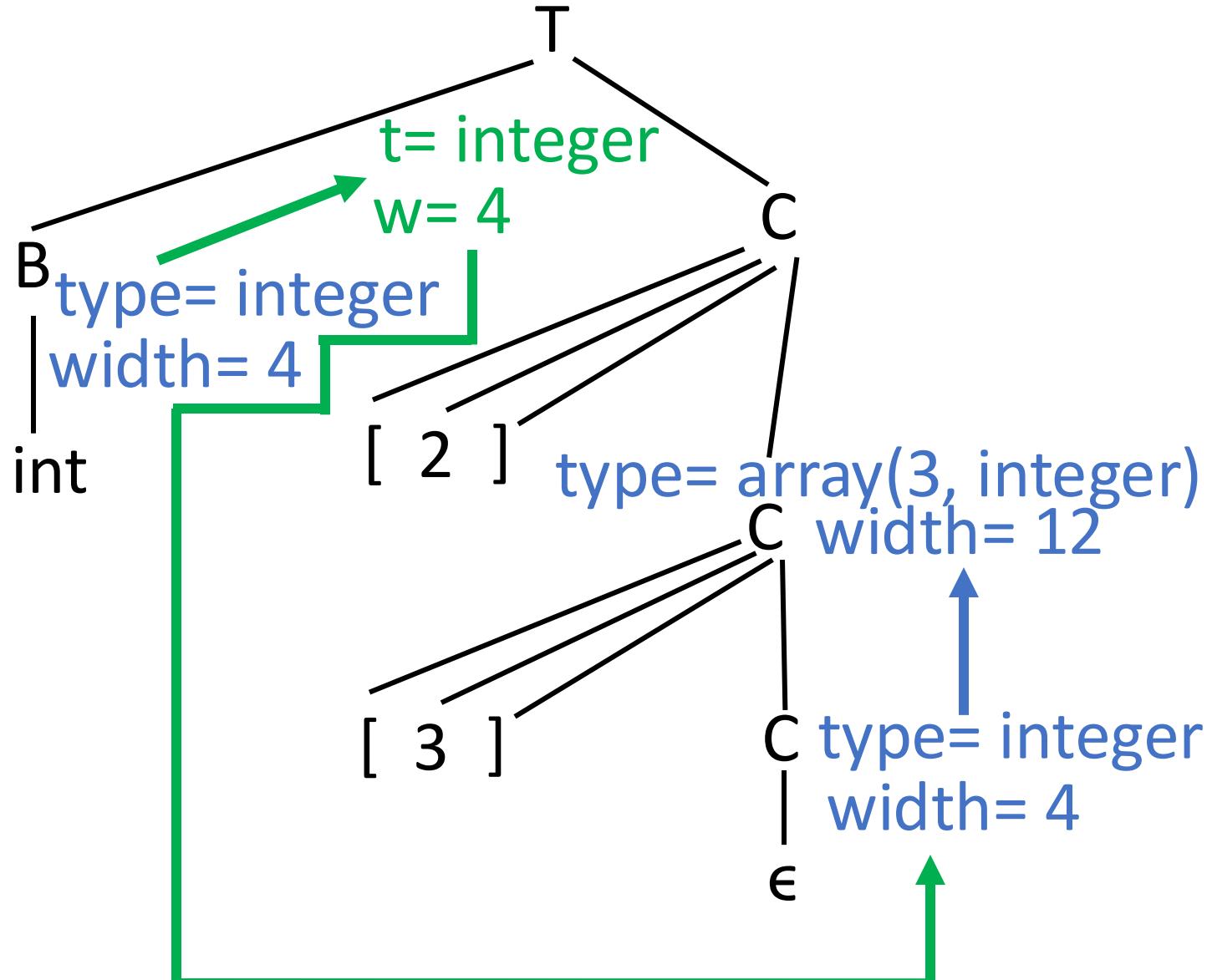


## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

T → B       $\{ t = B.type; w = B.width; \}$   
C       $\{ T.type = C.type; T.width = C.width; \}$   
B → int       $\{ B.type = integer; B.width = 4; \}$   
B → float       $\{ B.type = float; B.width = 8; \}$   
C → ε       $\{ C.type = t; C.width = w; \}$   
C → [num] C<sub>1</sub>       $\{ C.type = array(num.value, C_1.type); C.width = num.value \times C_1.width; \}$

Computing types and their width

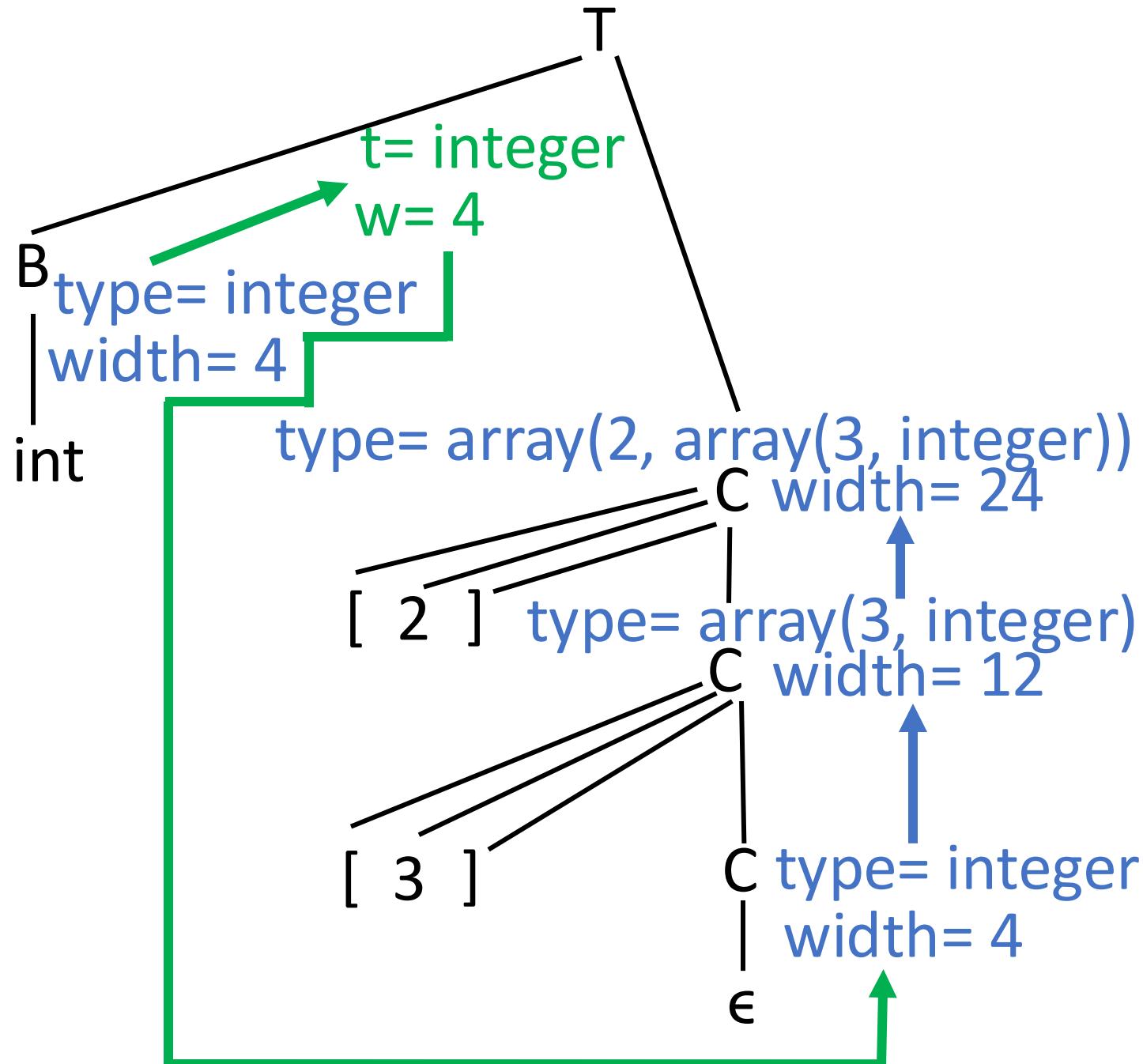


## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

T → B       $\{ t = B.type; w = B.width; \}$   
C       $\{ T.type = C.type; T.width = C.width; \}$   
B → int       $\{ B.type = integer; B.width = 4; \}$   
B → float       $\{ B.type = float; B.width = 8; \}$   
C → ε       $\{ C.type = t; C.width = w; \}$   
C → [num] C<sub>1</sub>       $\{ C.type = array(num.value, C_1.type); C.width = num.value \times C_1.width; \}$

Computing types and their width

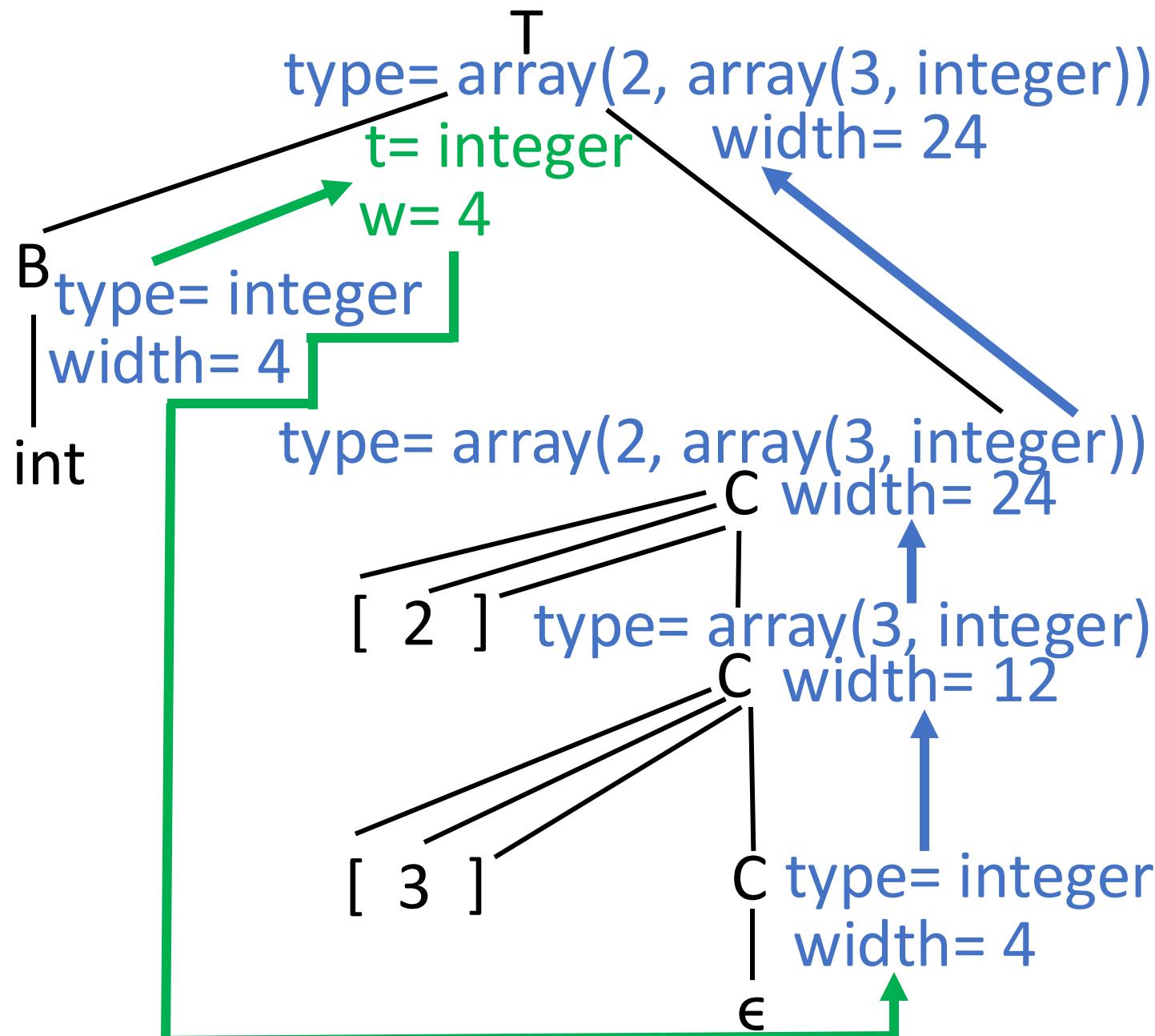


## Example 6.9 (Cont...)

### SDT of Array Types for int [2] [3]

T → B      { t = B.type; w = B.width; }  
C      { T.type = C.type; T.width = C.width; }  
B → int      { B.type = integer; B.width = 4; }  
B → float      { B.type = float; B.width = 8; }  
C → ε      { C.type = t; C.width = w; }  
C → [num] C<sub>1</sub> { C.type = array(num.value, C<sub>1</sub>.type);  
                  C.width = num.value X C<sub>1</sub>.width; }

Computing types and their width



Syntax-directed translation of array types

# Comparison of Example 6.9 and Example 5.13

$T \rightarrow B$	{ $t = B.type; w = B.width;$ }
$C$	{ $T.type = C.type; T.width = C.width;$ }
$B \rightarrow \text{int}$	{ $B.type = \text{integer}; B.width = 4;$ }
$B \rightarrow \text{float}$	{ $B.type = \text{float}; B.width = 8;$ }
$C \rightarrow \epsilon$	{ $C.type = t; C.width = w;$ }
$C \rightarrow [\text{num}] C_1$	{ $C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width;$ }

Computing types and their width

PRODUCTION	SEMANTIC RULES
$T \rightarrow B C$	$T.type = C.type$ $T.width = C.width$
$B \rightarrow \text{int}$	$B.type = \text{integer}$
$B \rightarrow \text{float}$	$B.type = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.type = \text{array}(\text{num.value}, C_1.type)$ $C.width = \text{num.value} \times C_1.width$
$C \rightarrow \epsilon$	$C.type = \epsilon.type$ $C.width = \epsilon.width$

$T$  generates either a basic type or an array type

# Sequences of Declarations

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group
- We can use a variable say offset to keep track of the next available relative address
- The translation scheme of figure deals with a sequence of declarations of the form  $T \text{ id}$  where  $T$  generates a type as in previous figure

$$\begin{array}{ll} P \rightarrow & \{ \text{offset} = 0; \} \\ & D \\ D \rightarrow T \text{ id} ; & \{ \text{top.put(id.lexeme, } T.\text{type, offset}); \\ & \quad \text{offset} = \text{offset} + T.\text{width}; \} \\ & D_1 \\ D \rightarrow \epsilon & \end{array}$$

Computing the relative addresses of declared names

# Sequences of Declarations (Cont...)

- The initialization of offset is more evident if the first production appears on one line as:

P → {offset=0;} D

- Nonterminals generating  $\epsilon$ , called marker nonterminals can be used to rewrite productions so that all actions appear at the ends of right sides
- Using a marker nonterminal M the previous equation can be restated as:

P → M D

M →  $\epsilon$  {offset = 0;}

# Translation of Expressions

# Three-address code for expressions - SDD

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{id} = E ;$	$S.\text{code} = E.\text{code}   $ $\quad \text{gen}(\text{top.get(id.lexeme)} ' =' E.\text{addr})$
$E \rightarrow E_1 + E_2$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code}    E_2.\text{code}   $ $\quad \text{gen}(E.\text{addr} ' =' E_1.\text{addr} ' +' E_2.\text{addr})$
$  - E_1$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code}   $ $\quad \text{gen}(E.\text{addr} ' =' \text{'minus'} E_1.\text{addr})$
$  ( E_1 )$	$E.\text{addr} = E_1.\text{addr}$ $E.\text{code} = E_1.\text{code}$
$  \text{id}$	$E.\text{addr} = \text{top.get(id.lexeme)}$ $E.\text{code} = ''$

Task-1: Draw  
Annotated parse  
tree for  $a = b + -c$   
and find  $S.\text{code}$

# Incremental Translation - SDT

$S \rightarrow \mathbf{id} = E ; \quad \{ \text{gen}( \text{top.get(id.lexeme)} '==' E.\text{addr}); \}$

$E \rightarrow E_1 + E_2 \quad \{ E.\text{addr} = \mathbf{new} \text{ Temp}();$   
 $\quad \quad \quad \text{gen}(E.\text{addr} '==' E_1.\text{addr} '+' E_2.\text{addr}); \}$

|  $- E_1 \quad \{ E.\text{addr} = \mathbf{new} \text{ Temp}();$   
 $\quad \quad \quad \text{gen}(E.\text{addr} '==' '\mathbf{minus}' E_1.\text{addr}); \}$

|  $( E_1 ) \quad \{ E.\text{addr} = E_1.\text{addr}; \}$

|  $\mathbf{id} \quad \{ E.\text{addr} = \text{top.get(id.lexeme)}; \}$

Task-2: Draw  
Annotated parse  
tree for  $\mathbf{a} = \mathbf{b} + -\mathbf{c}$   
and output the  
three address  
code.

# Translation of Array References - SDT

$S \rightarrow \text{id} = E ; \quad \{ \text{gen}(\text{top.get(id.lexeme)} '==' E.\text{addr}); \}$

$| \quad L = E ; \quad \{ \text{gen}(L.\text{array.base}'[' L.\text{addr} ']' '==' E.\text{addr}); \}$

$E \rightarrow E_1 + E_2 \quad \{ E.\text{addr} = \text{new Temp}();$   
 $\quad \quad \quad \text{gen}(E.\text{addr} '==' E_1.\text{addr} +' E_2.\text{addr}); \}$

$| \quad \text{id} \quad \quad \quad \{ E.\text{addr} = \text{top.get(id.lexeme)}; \}$

$| \quad L \quad \quad \quad \{ E.\text{addr} = \text{new Temp}();$   
 $\quad \quad \quad \text{gen}(E.\text{addr} '==' L.\text{array.base}'[' L.\text{addr} ']'); \}$

$L \rightarrow \text{id} [ E ] \quad \{ L.\text{array} = \text{top.get(id.lexeme)};$   
 $\quad \quad \quad L.\text{type} = L.\text{array.type.elem};$   
 $\quad \quad \quad L.\text{addr} = \text{new Temp}();$   
 $\quad \quad \quad \text{gen}(L.\text{addr} '==' E.\text{addr} '*' L.\text{type.width}); \}$

$| \quad L_1 [ E ] \quad \{ L.\text{array} = L_1.\text{array};$   
 $\quad \quad \quad L.\text{type} = L_1.\text{type.elem};$   
 $\quad \quad \quad t = \text{new Temp}();$   
 $\quad \quad \quad L.\text{addr} = \text{new Temp}();$   
 $\quad \quad \quad \text{gen}(t '==' E.\text{addr} '*' L.\text{type.width});$   
 $\quad \quad \quad \text{gen}(L.\text{addr} '==' L_1.\text{addr} +' t); \}$

**Task-3:** Draw Annotated parse tree for C + A[i][j] and Three-Address code considering A is an array declaration of 2X3.

# Practice

- Use the Translation of Array References for the following assignment to generate Three-address code.
  - $X = a[i] + b[j]$
  - $X = a[i][j] + b[i][j]$

End

# **Chapter-07**

# **Run-Time Environments**

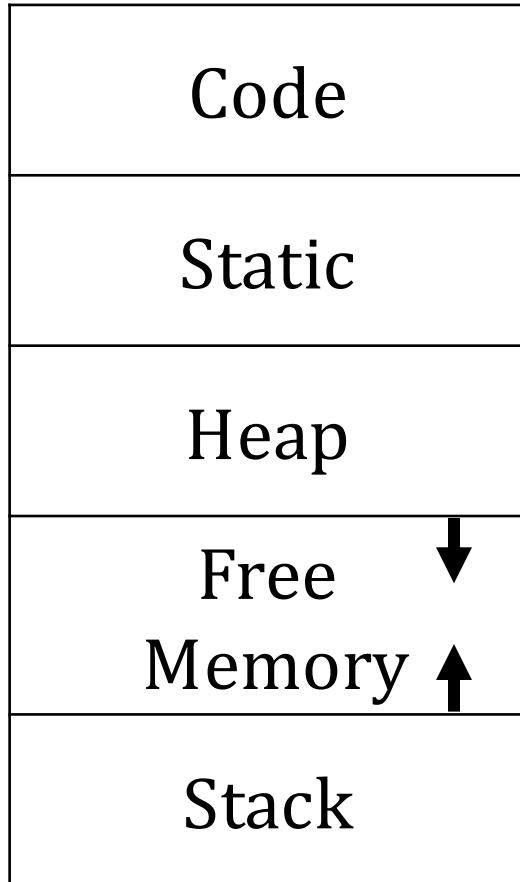
# Outline

- ❑ Storage Organization
  - ❑ Static Versus Dynamic Storage Allocation
- ❑ Stack Allocation of Space
  - ❑ Activation Trees
  - ❑ Activation Records
  - ❑ Calling Sequences
  - ❑ Variable-Length Data on the Stack

# Storage Organization

- ❑ From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location
- ❑ The management and organization of this logical address space is shared between the compiler, operating system and target machine
- ❑ The operating system maps the logical addresses into physical addresses which are usually spread throughout memory

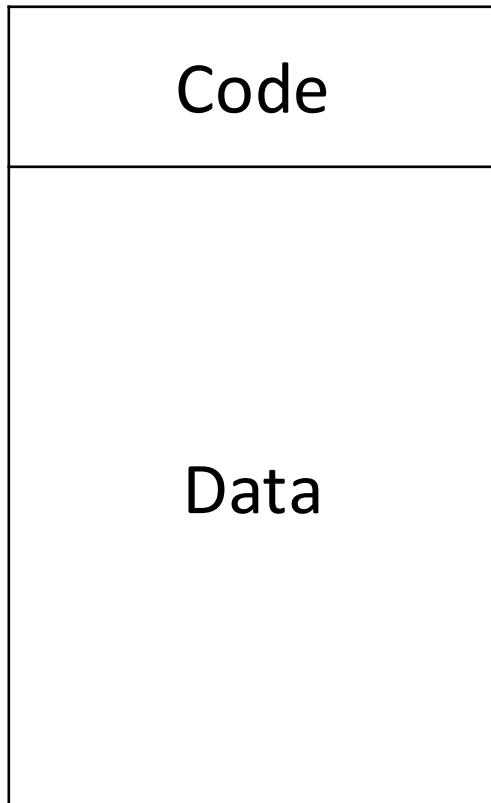
# Storage Organization (Cont...)



- The run-time representation of an object program in the logical address space consists of data and program areas as shown in figure
- A compiler for a language like **C++** on an operating system like **Linux** might subdivide memory in this way

**Typical Subdivision of Runtime Memory into Code and Data Areas**

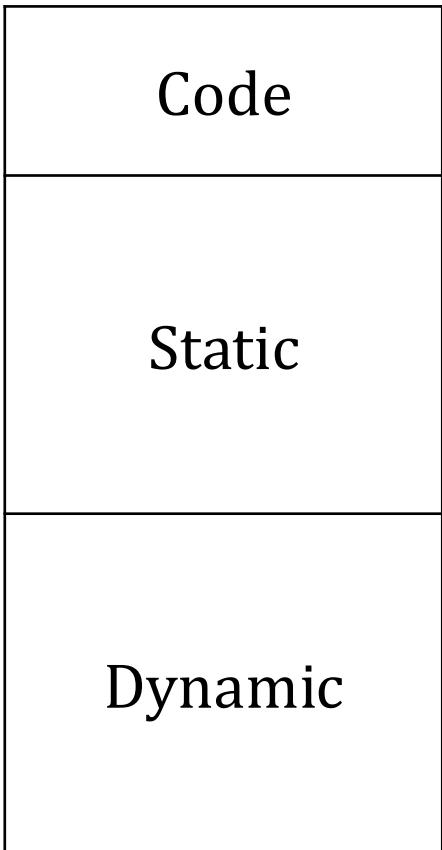
# Storage Organization (Cont...)



- ❖ Runtime Memory is first subdivided into two areas: **Code** and **Data** Areas
- ❖ The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area **Code** usually in the low end of memory
- ❖ Data area is further subdivided into two areas: **Static** and **Dynamic** areas

**Typical Subdivision of Runtime Memory into Code and Data Areas**

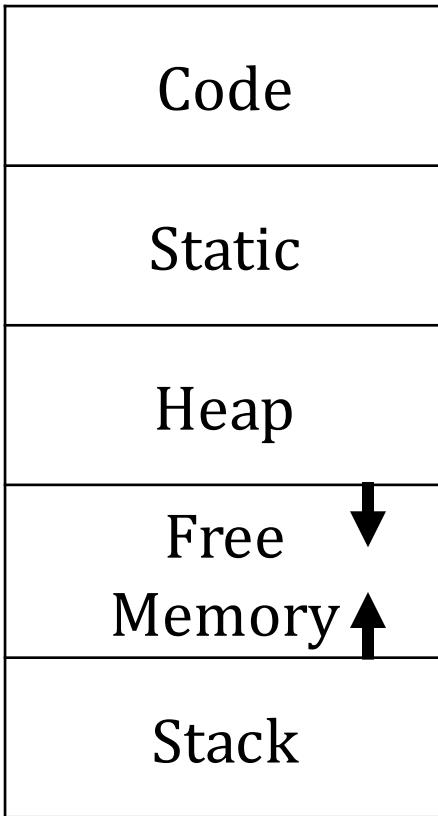
# Storage Organization (Cont...)



- ❖ The size of some program data objects such as global constants and data generated by the compiler such as information to support garbage collection may be known at compile time and these data objects can be placed in another statically determined area called **Static**
- ❖ Dynamic Data area is further subdivided into two areas: **Stack** and **Heap** areas

**Typical Subdivision of Runtime Memory into Code and Data Areas**

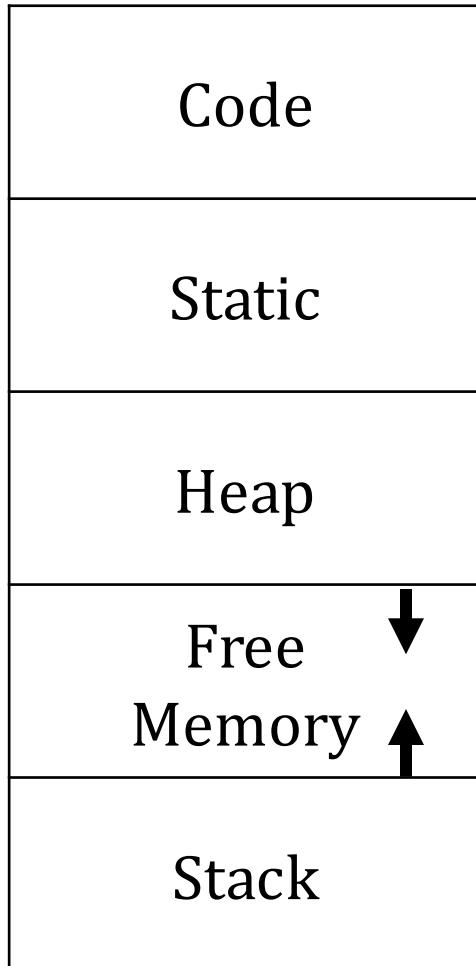
## Storage Organization (Cont...)



- To maximize the utilization of space at run time the other two areas **Stack** and **Heap** are at the opposite ends of the remainder of the address space
- These areas are dynamic
- Their size can change as the program executes
- These areas grow towards each other as needed

**Typical Subdivision of Runtime Memory into Code and Data Areas**

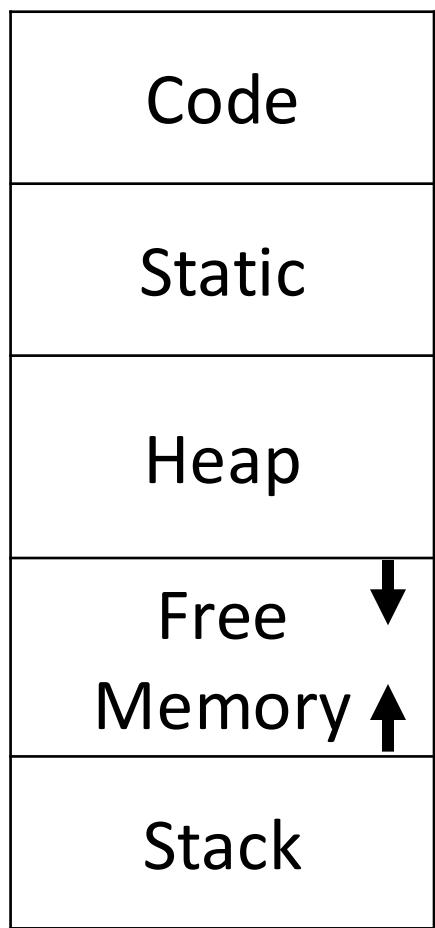
# Storage Organization (Cont...)



- The **stack** is used to store data structures called activation records that get generated during procedure calls
- In practice, the **stack** grows towards lower addresses and the **heap** towards higher
- However, in this book the authors assume that the stack grows towards higher addresses so that they can use positive offsets for examples

**Typical Subdivision of Runtime Memory into Code and Data Areas**

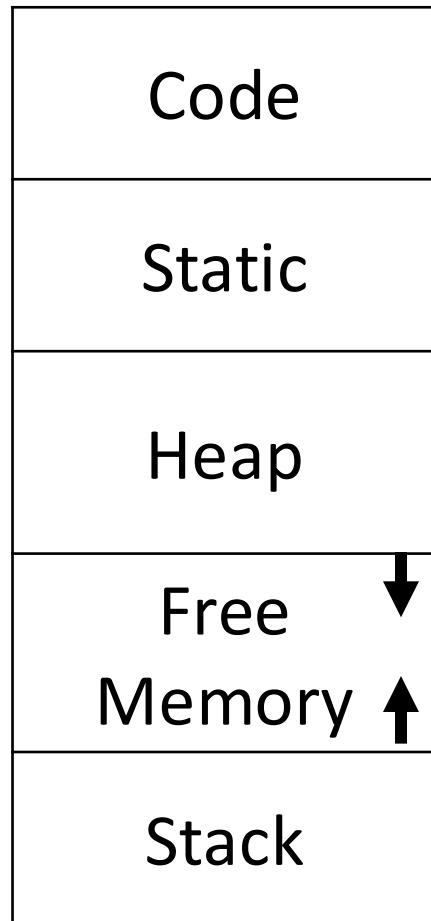
# Storage Organization (Cont...)



- An activation record is used to store information about the status of the machine such as the value of the program counter and machine registers when a procedure call occurs
- When control returns from the call the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call
- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation

**Typical Subdivision of Runtime Memory into Code and Data Areas**

# Storage Organization (Cont...)



- Many programming languages allow the programmer to allocate and deallocate data under program control
- For example, C has the functions malloc and free that can be used to obtain and give back arbitrary chunks of storage
- The **heap** is used to manage this kind of dynamic data

**Typical Subdivision of Runtime Memory into Code and Data Areas**

# Padding

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine
- On many machines instructions to add integers may expect integers to be aligned that is placed at an address divisible by 4
- Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment, leaving 2 bytes unused
- Space left unused due to alignment considerations is referred to as **padding**
- When space is at a premium, a compiler may pack data so that no padding is left. Additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned

# Static Versus Dynamic Storage Allocation

- The layout and allocation of data to memory locations in the run-time environment are key issues in storage management
- The two adjectives static and dynamic distinguish between compile time and run time respectively
- We say that a storage-allocation decision is static if it can be made by the compiler looking only at the text of the program not at what the program does when it executes
- Conversely, a decision is dynamic if it can be decided only while the program is running

# Static Versus Dynamic Storage Allocation (Cont...)

- Many compilers use some combination of the following two strategies for dynamic storage allocation:
  - **Stack storage:** Names local to a procedure are allocated space on a stack
  - **Heap storage:** Data that may outlive the call to the procedure that created it is usually allocated on a “heap” of reusable storage

# Stack Allocation of Space

- Almost all compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack
- Each time a procedure is called space for its local variables is pushed onto a stack and when the procedure terminates that space is popped off the stack
- This arrangement allows space to be shared by procedure calls whose durations do not overlap in time
- It allows to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same regardless of the sequence of procedure calls

# Activation Trees

Stack allocation would not be feasible if procedure calls, or activations of procedures did not nest in time

## Example 7.1

- Figure contains a sketch of a program that reads nine integers into an array  $a$  and sorts them using the recursive quicksort algorithm
- The main function has three tasks
- It calls `readArray`, sets the sentinels, and then calls `quicksort` on the entire data array

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}

int partition(int m, int n) {
    /* Picks a separator value  $v$ , and partitions  $a[m..n]$  so that
        $a[m..p-1]$  are less than  $v$ ,  $a[p] = v$ , and  $a[p+1..n]$  are
       equal to or greater than  $v$ . Returns  $p$ . */
    ...
}

void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

## Example 7.1 (Cont...)

- This program uses two auxiliary functions `readArray` and `partition`
- The function `readArray` is used only to load the data into the array `a`
- The first and last elements of `a` are used for sentinels set in the main function
- We assume `a[0]` and `a[10]` are set to a value lower than and higher than any possible data value respectively

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}

int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p - 1] are less than v, a[p] = v, and a[p + 1..n] are
       equal to or greater than v. Returns p. */
    ...
}

void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

## Example 7.1 (Cont...)

- The partition function divides a portion of the array, delimited by the arguments m and n
- So the low elements of  $a[m]$  through  $a[n]$  are at the beginning and the high elements are at the end
- Although neither group is necessarily in the sorted order

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}

int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p - 1] are less than v, a[p] = v, and a[p + 1..n] are
       equal to or greater than v. Returns p. */
    ...
}

void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

## Example 7.1 (Cont...)

- Quicksort procedure check if the condition  $n > m$  is true or not and if it is true then it calls partition function otherwise return
- Partition function returns an index  $i$  to separate the low and high elements
- These two groups of elements are then sorted by two recursive calls to quicksort

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}

int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p - 1] are less than v, a[p] = v, and a[p + 1..n] are
       equal to or greater than v. Returns p. */
    ...
}

void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

# Example 7.1 (Cont...)

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value  $v$ , and partitions  $a[m..n]$  so that
        $a[m..p-1]$  are less than  $v$ ,  $a[p] = v$ , and  $a[p+1..n]$  are
       equal to or greater than  $v$ . Returns  $p$ . */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
    enter partition (1,9) → p=4  
    leave partition (1,9)  
    enter quicksort (1,3)  
        ...  
        leave quicksort (1,3)  
    enter quicksort (5,9)  
        ...  
        leave quicksort (5,9)  
    leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Example 7.1 (Cont...)

- Figure suggests a sequence of calls that might result from an execution of the program
- In this execution, the call to `partition(1,9)` returns 4, so  $a[1]$  through  $a[3]$  hold elements less than its chosen separator value  $v$ , while the larger elements are in  $a[5]$  through  $a[9]$

```
enter main ()  
  enter readArray ()  
  leave readArray ()  
  enter quicksort (1,9)  
    enter partition (1,9) → p=4  
    leave partition (1,9)  
    enter quicksort (1,3)  
    ...  
    leave quicksort (1,3)  
    enter quicksort (5,9)  
    ...  
    leave quicksort (5,9)  
  leave quicksort (1,9)  
leave main ()
```

Possible Activations for the Quicksort Program

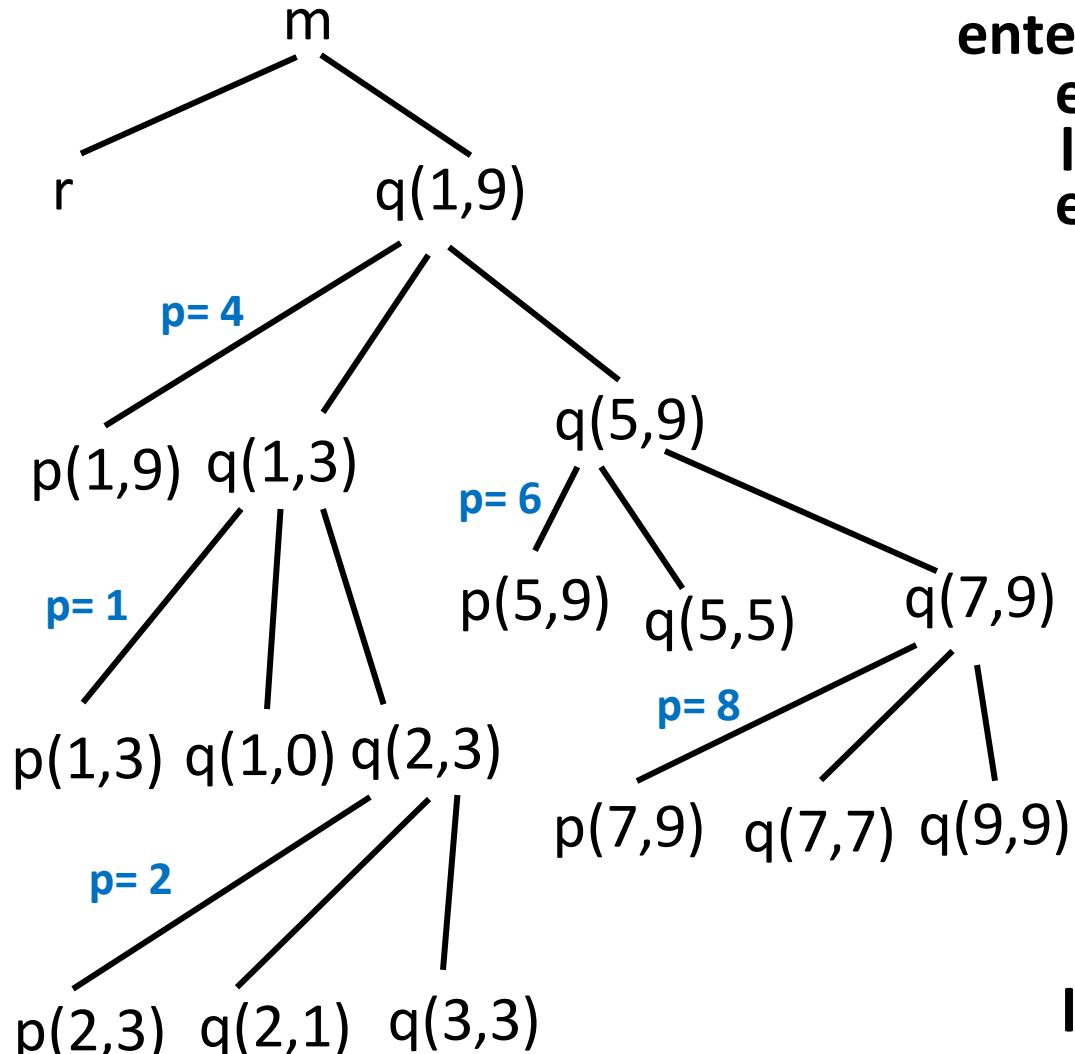
## Activation Trees (Cont...)

- In this example, as is true in general procedure activations are nested in time
- If an activation of procedure p calls procedure q then that activation of q must end before the activation of p can end
- There are three common cases:
  - **The activation of q terminates normally**
  - **The activation of q or some procedure q called either directly or indirectly aborts**
  - **The activation of q terminates because of an exception that q cannot handle**

# Activation Trees (Cont...)

- We therefore can represent the activations of procedures during the running of an entire program by a tree, called an activation tree
- Each node corresponds to one activation and the root is the activation of the “main” procedure that initiates execution of the program
- At a node for an activation of procedure  $p$  the children correspond to activations of the procedures called by this activation of  $p$
- We show these activations in the order that they are called from left to right
- Notice that one child must finish before the activation to its right can begin

# Example 7.2



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) → p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

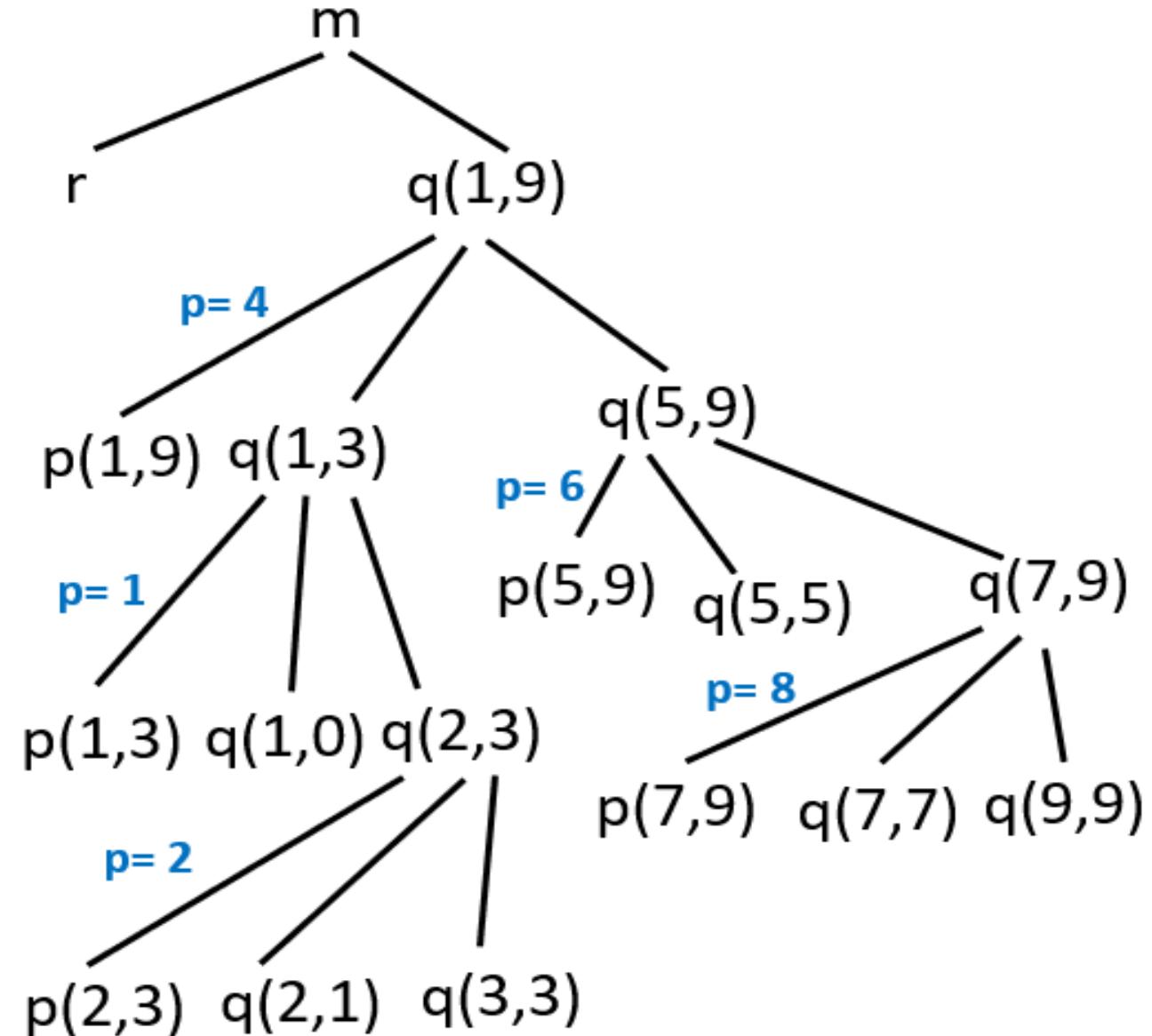
leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

- One possible activation tree that completes the sequence of calls and returns suggested is shown in figure
- Functions are represented by the first letters of their names
- Remember that this tree is only one possibility since the arguments of subsequent calls and also the number of calls along any branch is influenced by the values returned by partition



**Activation Tree Representing Calls  
During an Execution of Quicksort**

## Example 7.2 (Cont...)

m

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

    enter partition (1,9) → p= 4

    leave partition (1,9)

    enter quicksort (1,3)

        enter partition (1,3) → p= 1

        leave partition (1,3)

        enter quicksort (1,0)

        leave quicksort (1,0)

        enter quicksort (2,3)

            enter partition (2,3) → p= 2

            leave partition (2,3)

            enter quicksort (2,1)

            leave quicksort (2,1)

            enter quicksort (3,3)

            leave quicksort (3,3)

        leave quicksort (2,3)

    leave quicksort (1,3)

    enter quicksort (5,9)

        enter partition (5,9) → p= 6

        leave partition (5,9)

        enter quicksort (5,5)

        leave quicksort (5,5)

        enter quicksort (7,9) ↑ p= 8

            enter partition (7,9)

            leave partition (7,9)

            enter quicksort (7,7)

            leave quicksort (7,7)

            enter quicksort (9,9)

            leave quicksort (9,9)

        leave quicksort (7,9)

    leave quicksort (5,9)

        leave quicksort (1,9)

    leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

m  
r

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
    enter partition (1,9) → p= 4  
    leave partition (1,9)  
    enter quicksort (1,3)  
        enter partition (1,3) → p= 1  
        leave partition (1,3)  
        enter quicksort (1,0)  
        leave quicksort (1,0)  
        enter quicksort (2,3)  
            enter partition (2,3) → p= 2  
            leave partition (2,3)  
            enter quicksort (2,1)  
            leave quicksort (2,1)  
            enter quicksort (3,3)  
            leave quicksort (3,3)  
            leave quicksort (2,3)  
    leave quicksort (1,3)  
enter quicksort (5,9)  
    enter partition (5,9) → p= 6  
    leave partition (5,9)  
    enter quicksort (5,5)  
    leave quicksort (5,5)  
    enter quicksort (7,9) → p= 8  
        enter partition (7,9)  
        leave partition (7,9)  
        enter quicksort (7,7)  
        leave quicksort (7,7)  
        enter quicksort (9,9)  
        leave quicksort (9,9)  
        leave quicksort (7,9)  
    leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2 (Cont...)

m  
r

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

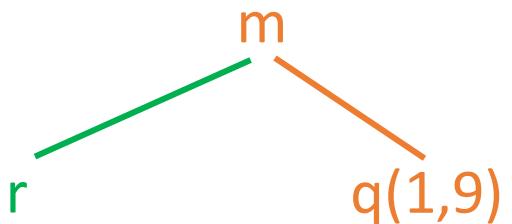
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2 (Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

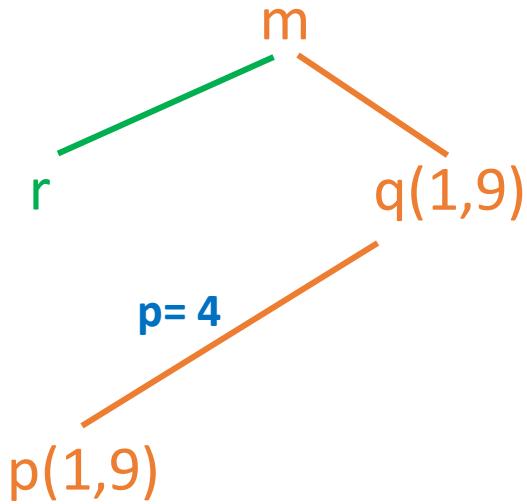
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) →  $p=4$

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) →  $p=1$

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) →  $p=2$

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) →  $p=6$

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑  $p=8$

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

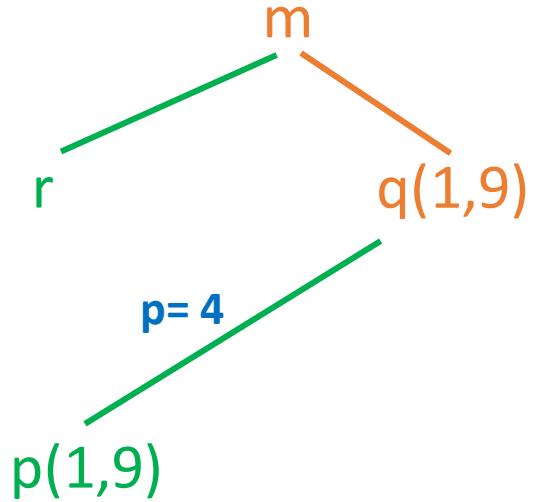
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4  
leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)  
leave quicksort (1,3)

enter quicksort (5,9)  
p= 1 enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9) p= 8

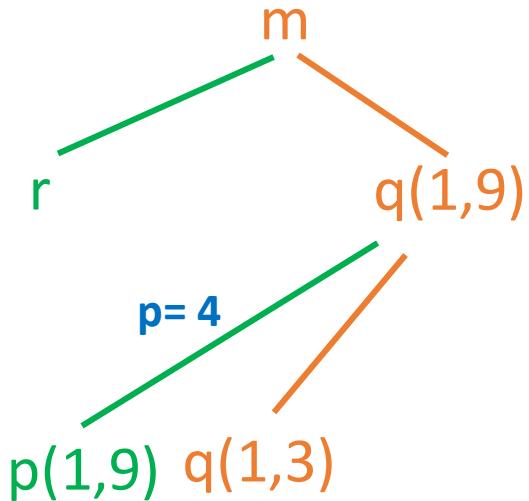
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)

leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

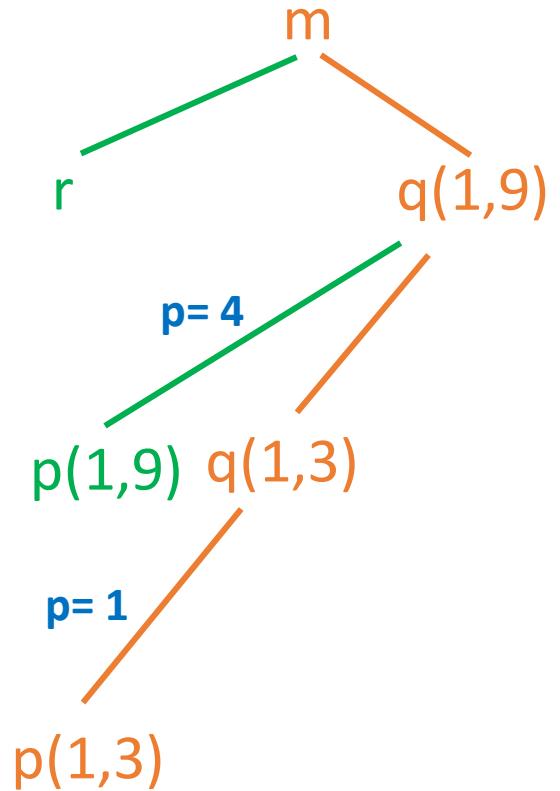
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

p= 1 enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) → p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

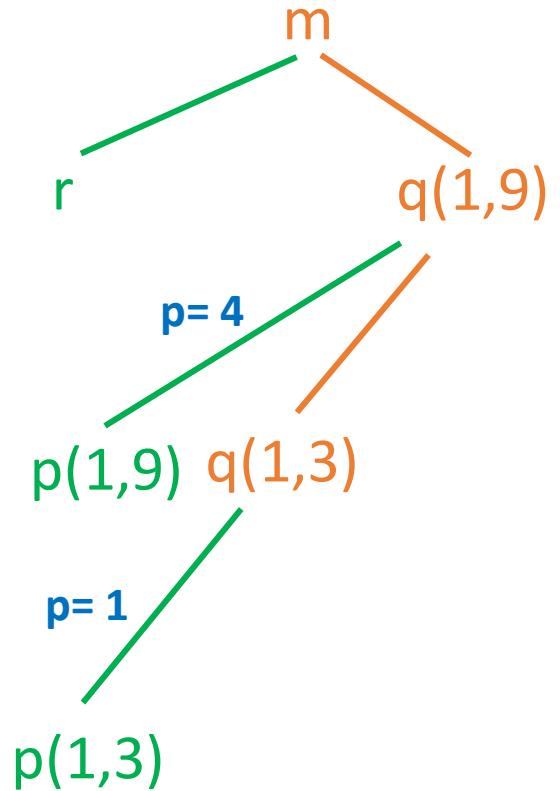
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)  
p= 1 enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9) p= 8

enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)

leave quicksort (7,9)

leave quicksort (5,9)

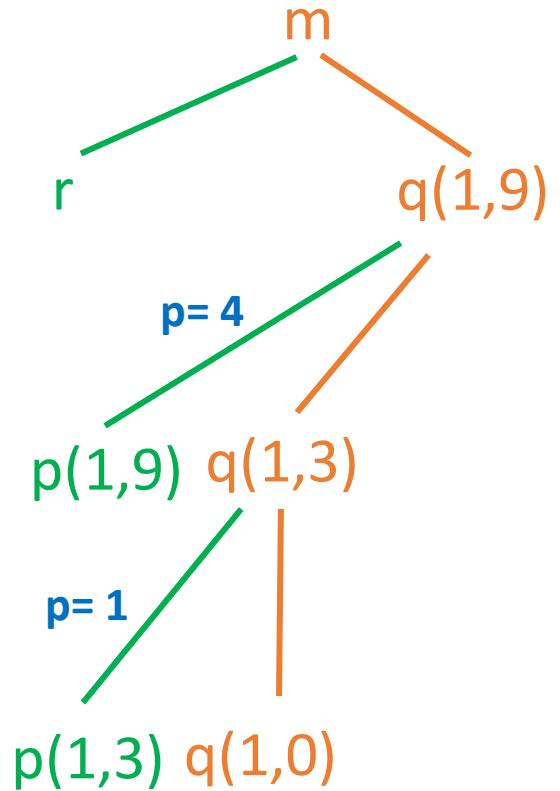
leave quicksort (1,9)

leave main ()

Activation Tree Representing Calls  
During an Execution of Quicksort

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)  
p= 1 enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

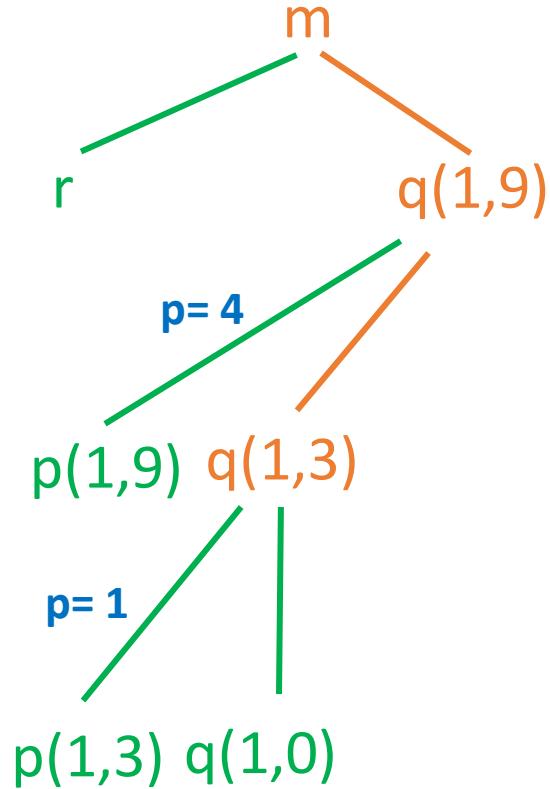
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

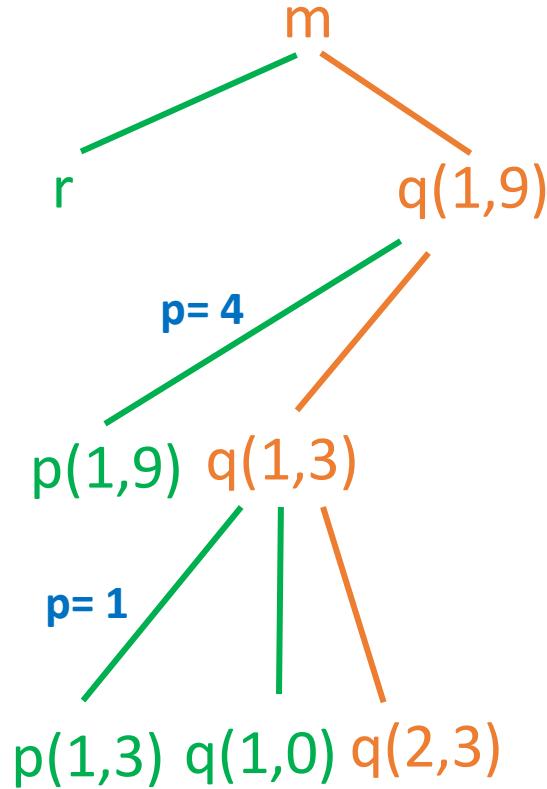


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

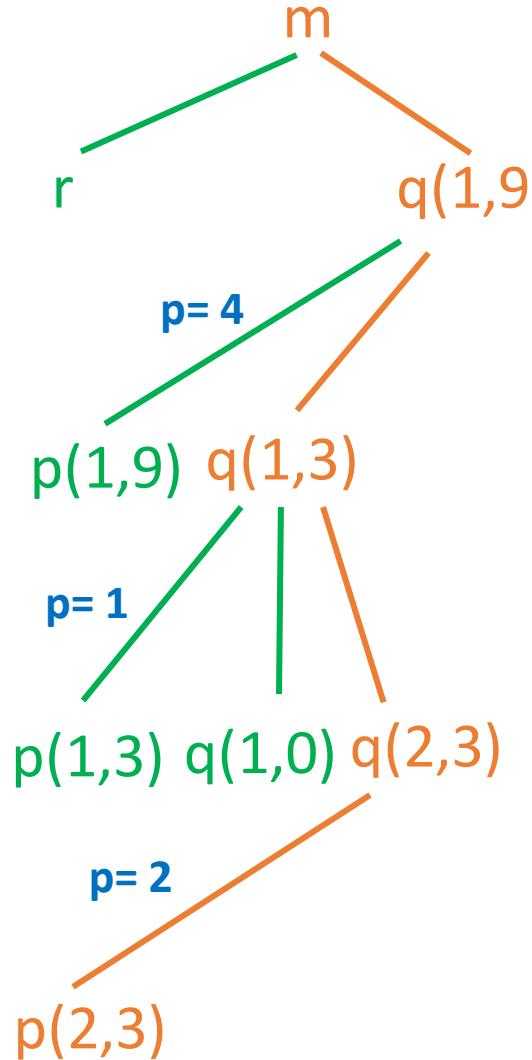


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

p= 1 enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

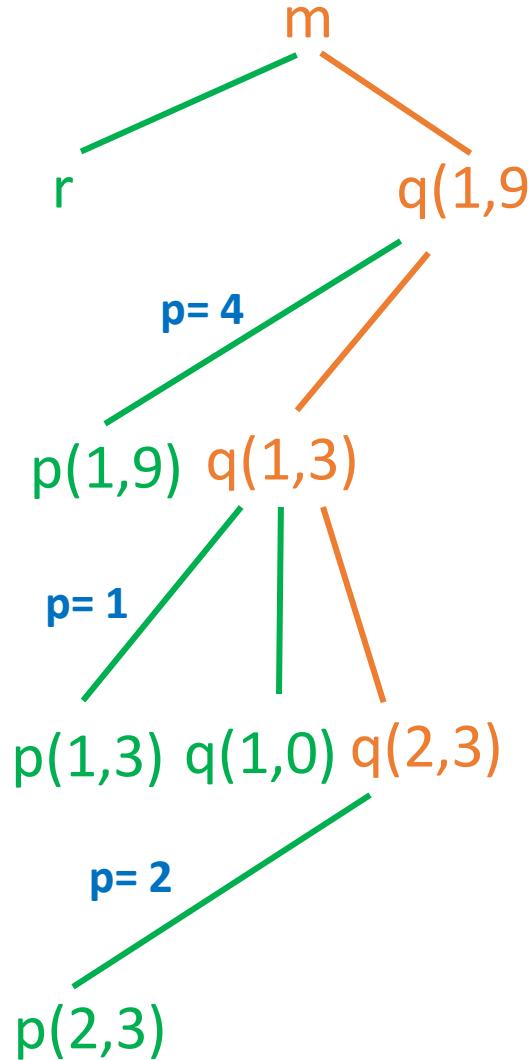
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

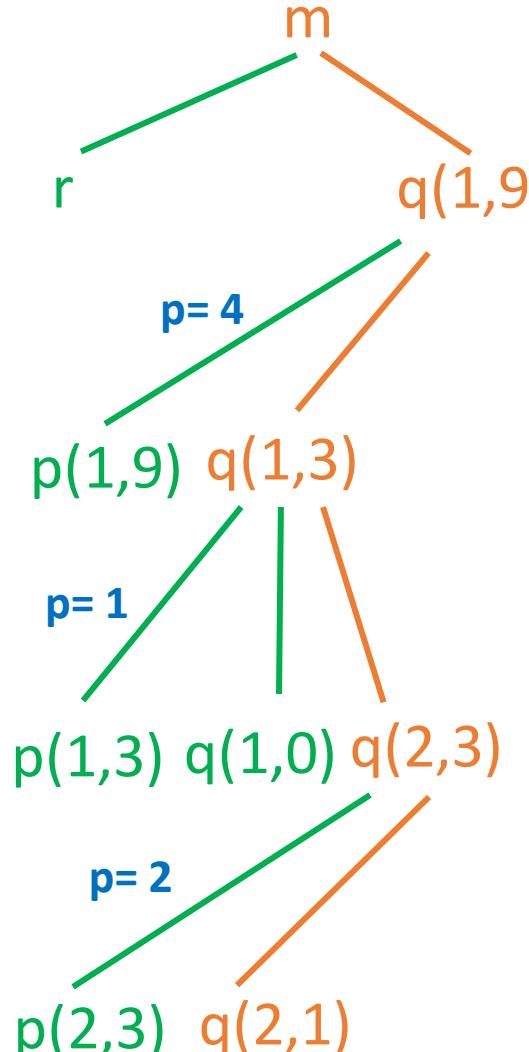


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

p= 1 enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

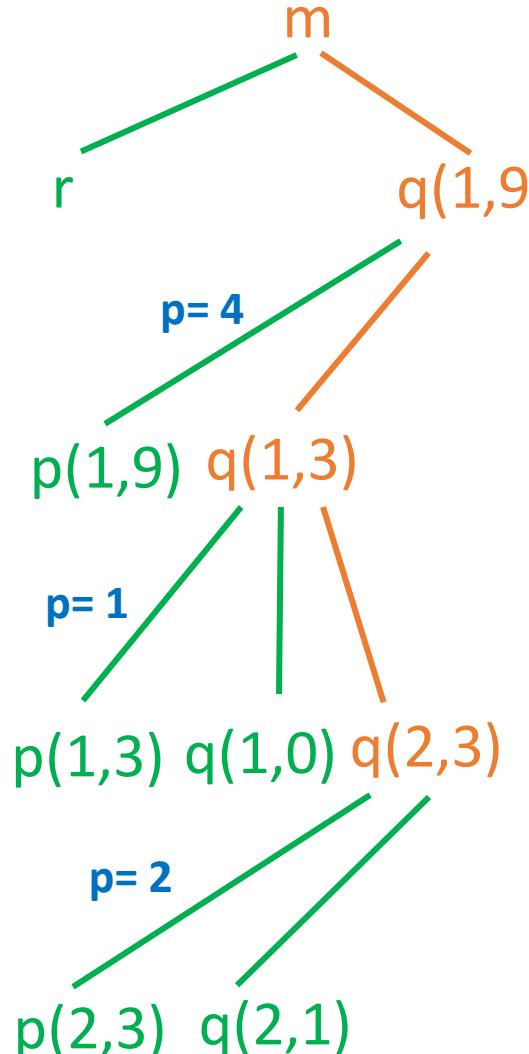
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

p= 1 enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

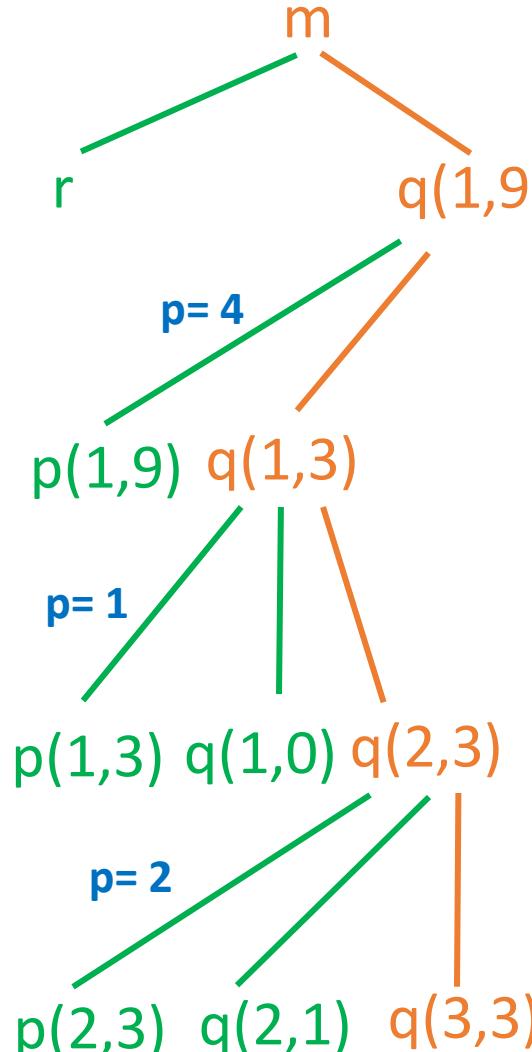
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

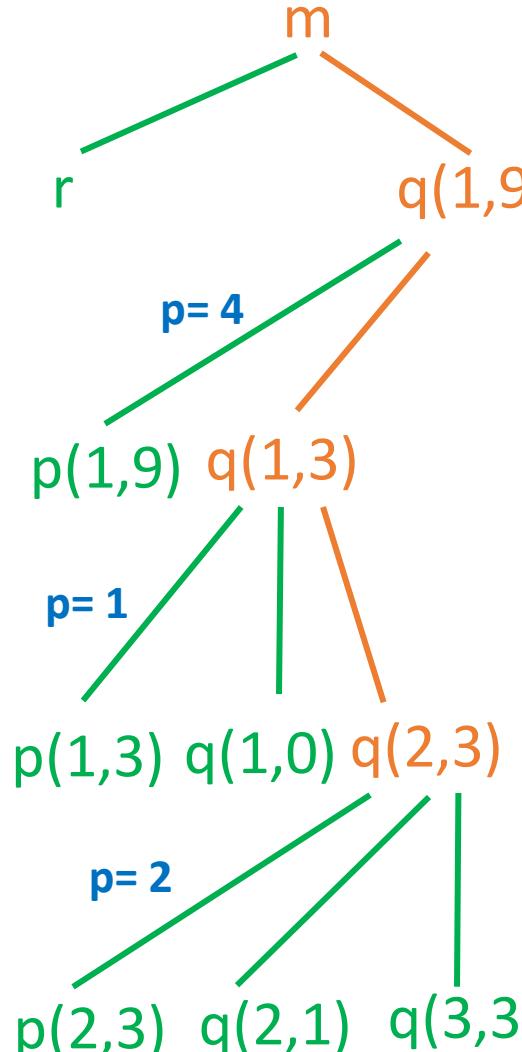


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) →  $p=4$

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) →  $p=1$

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) →  $p=2$

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

$p=1$  enter partition (5,9) →  $p=6$

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9)  $p=8$

$p=2$  enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

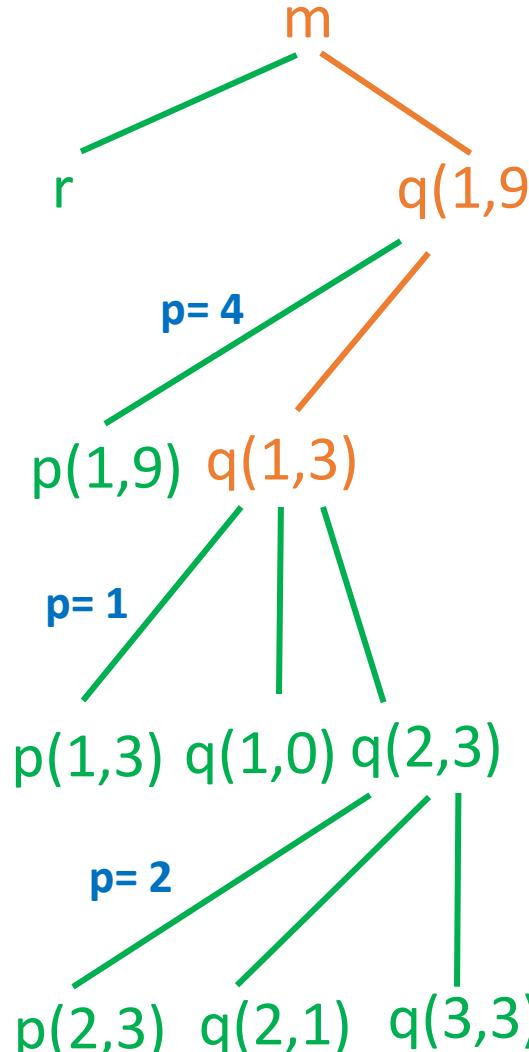
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

## Example 7.2(Cont...)

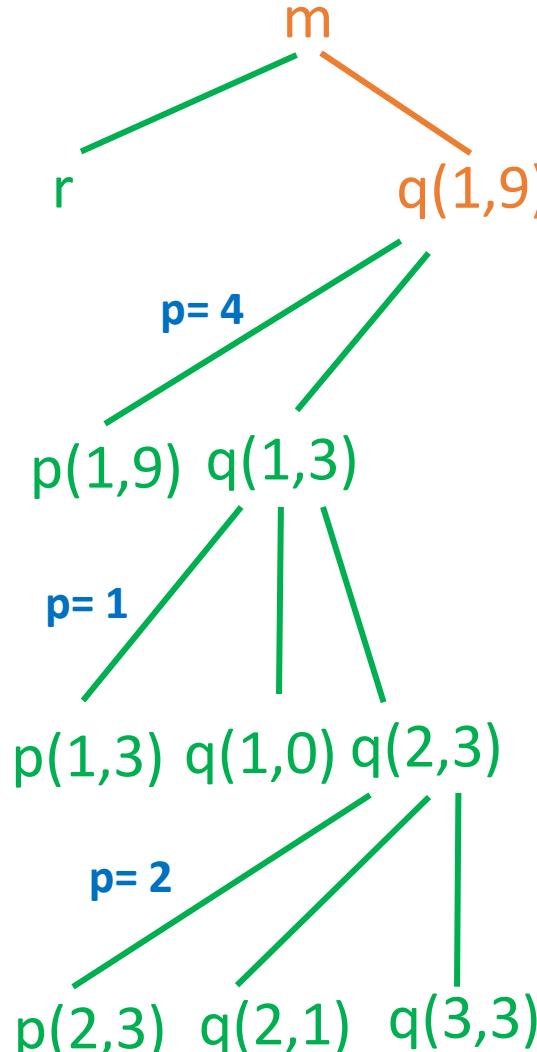


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

p= 1 enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

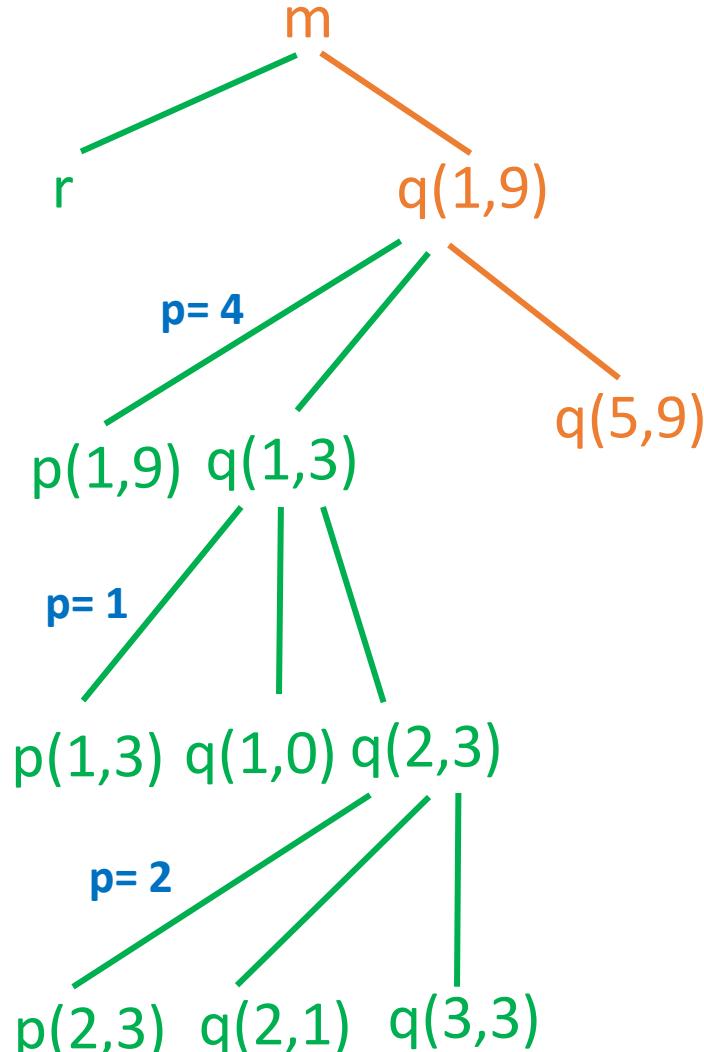
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

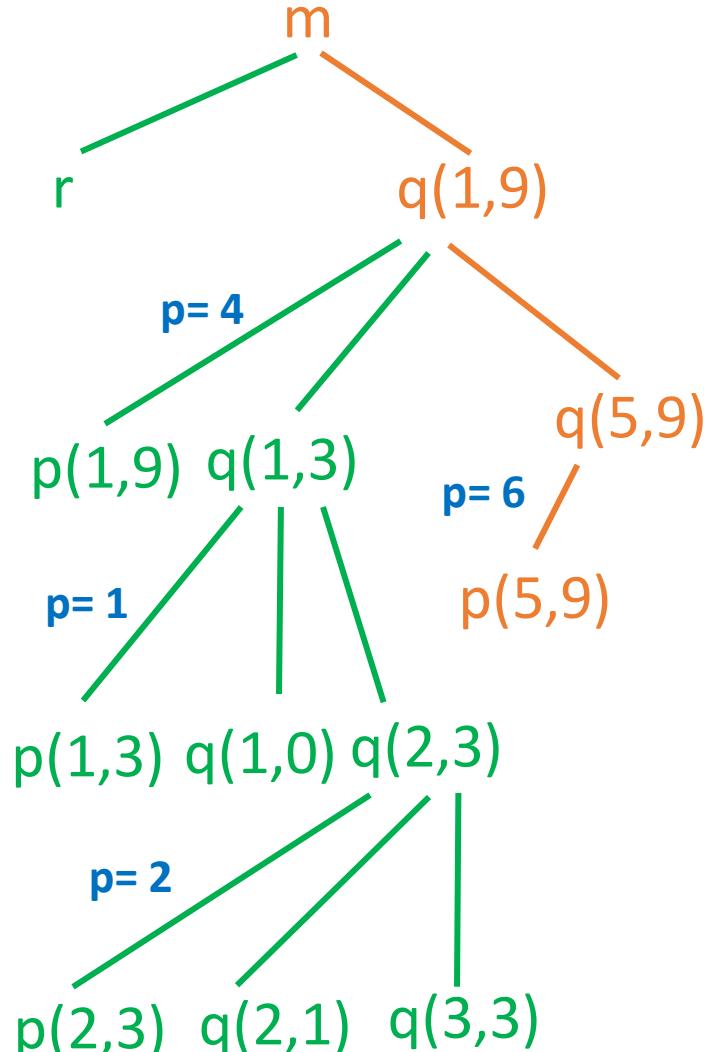


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

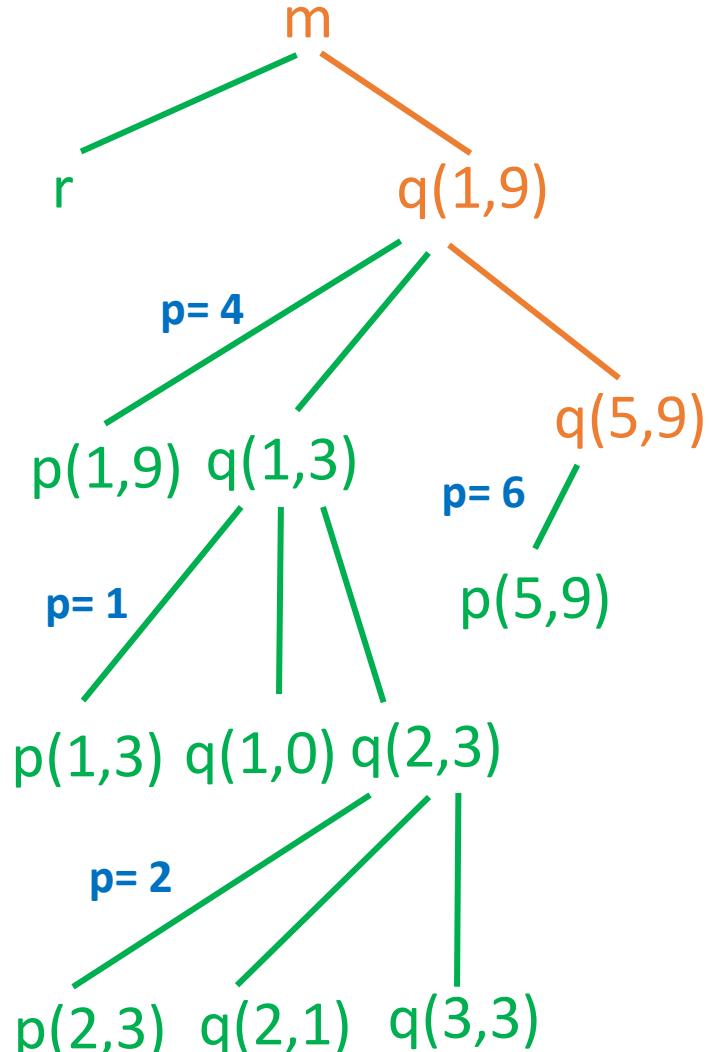


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

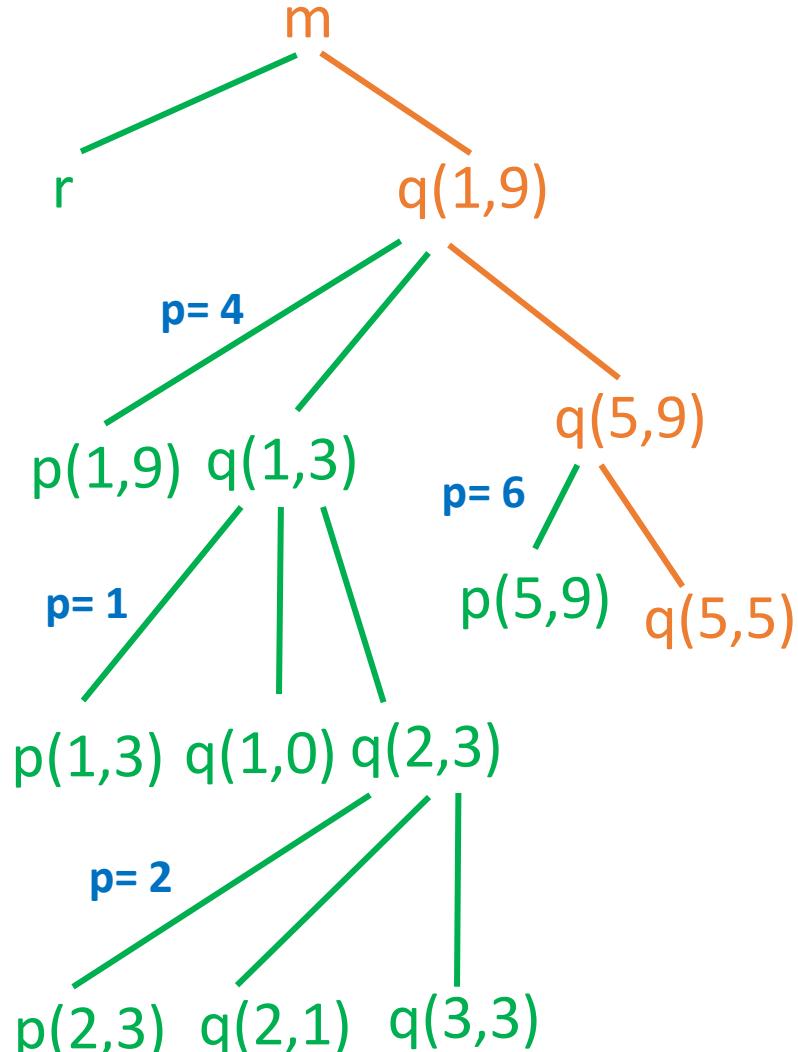


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

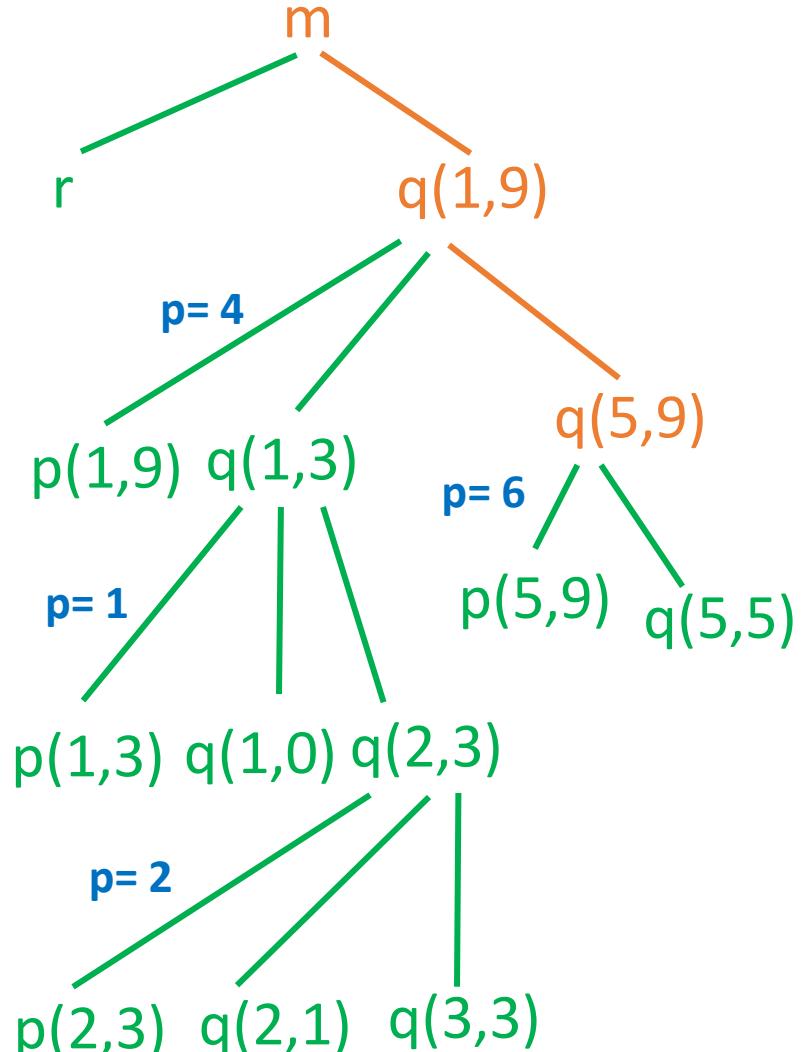


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9) → p= 8  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

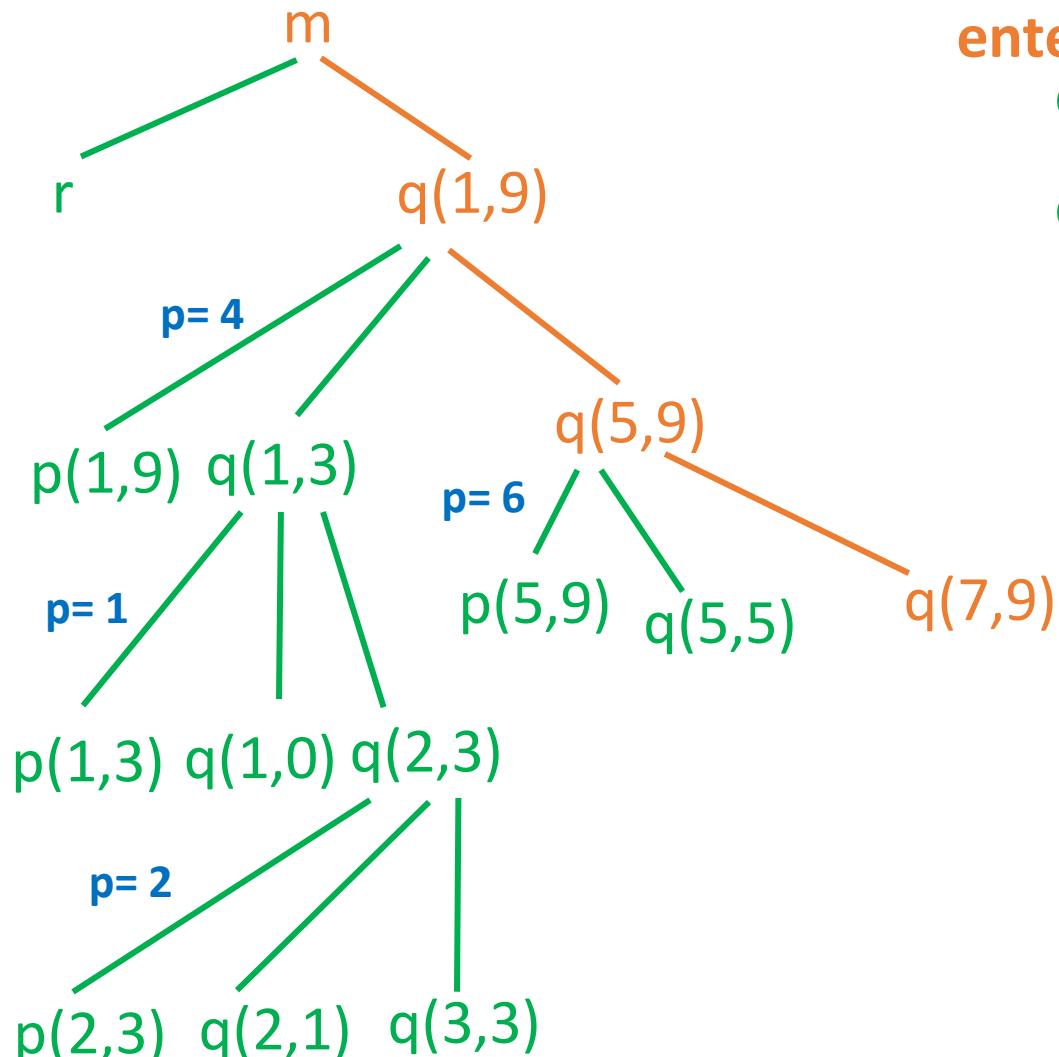


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

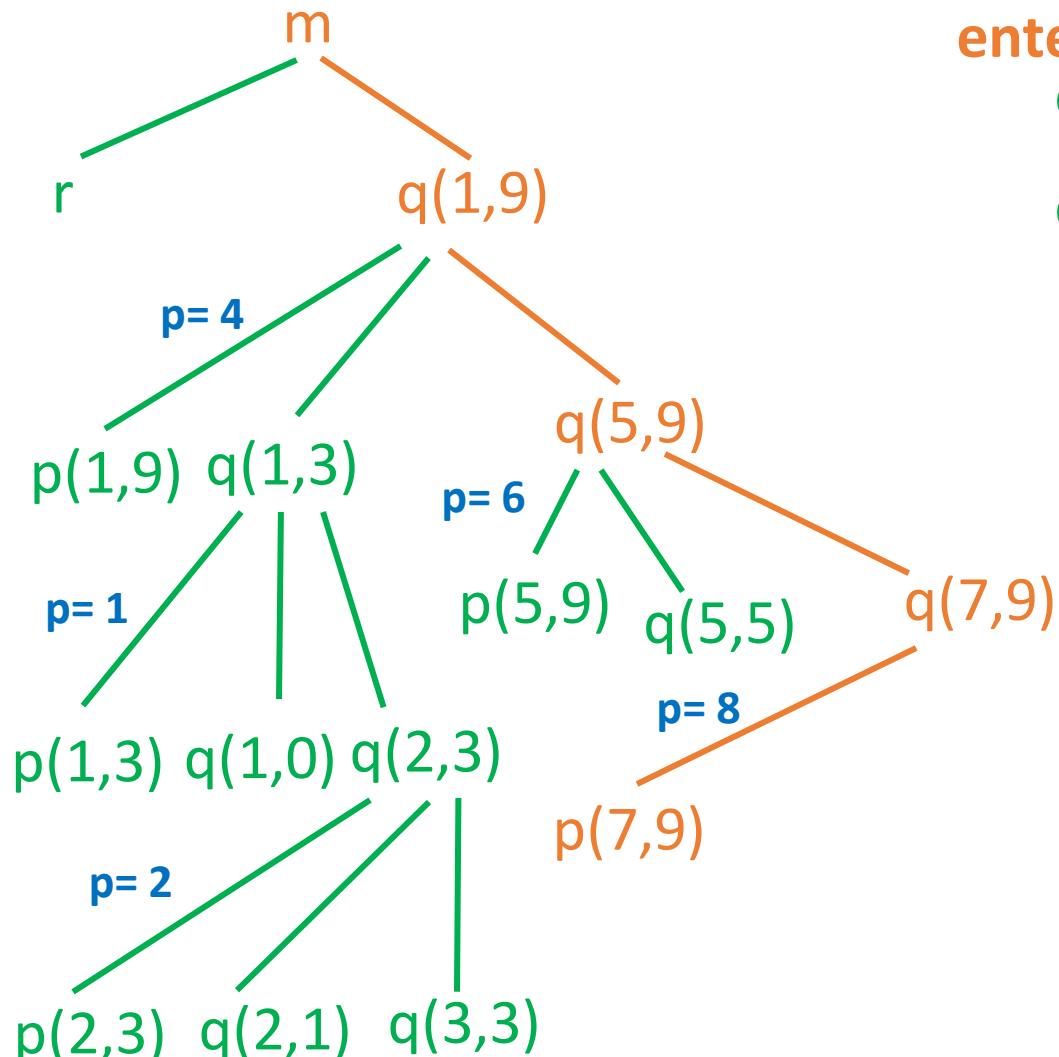


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9)  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

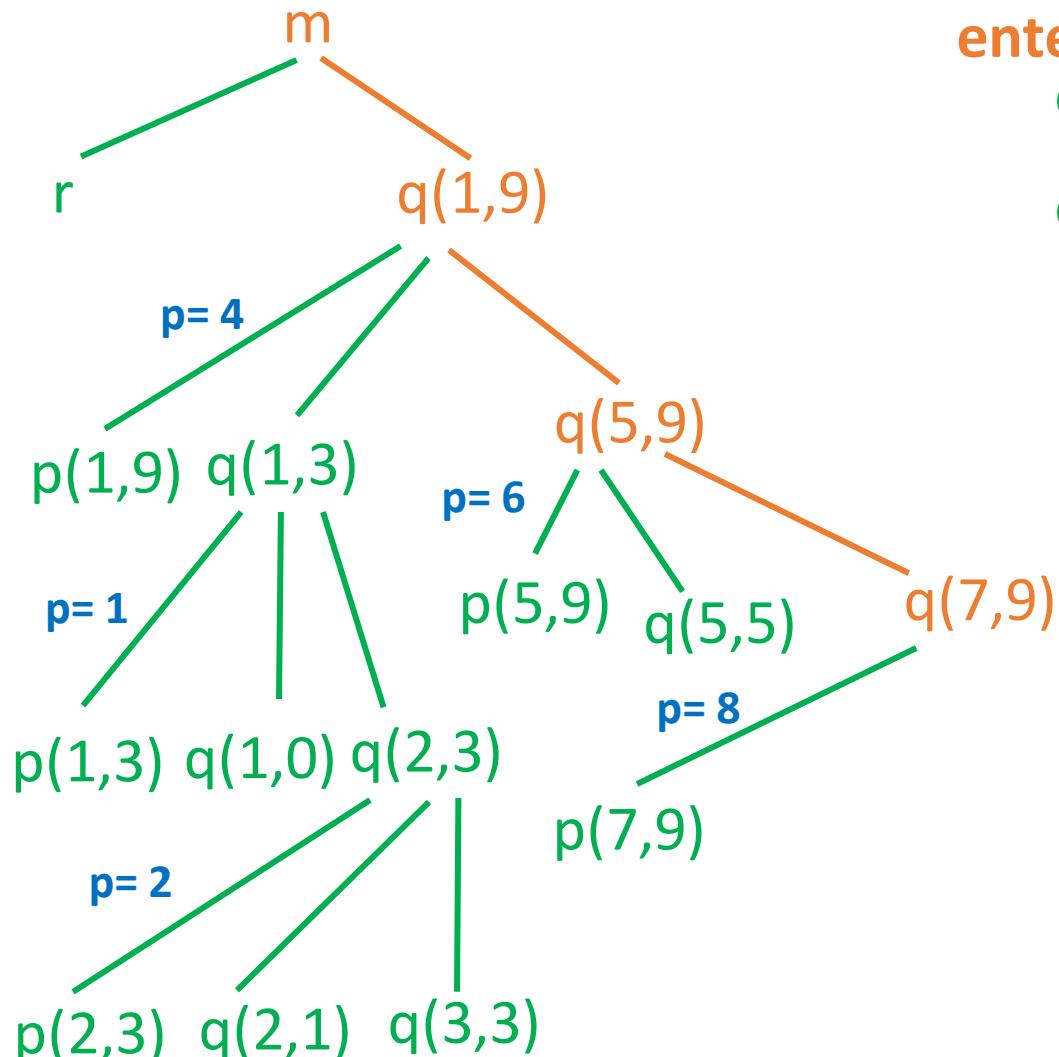
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9)

p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

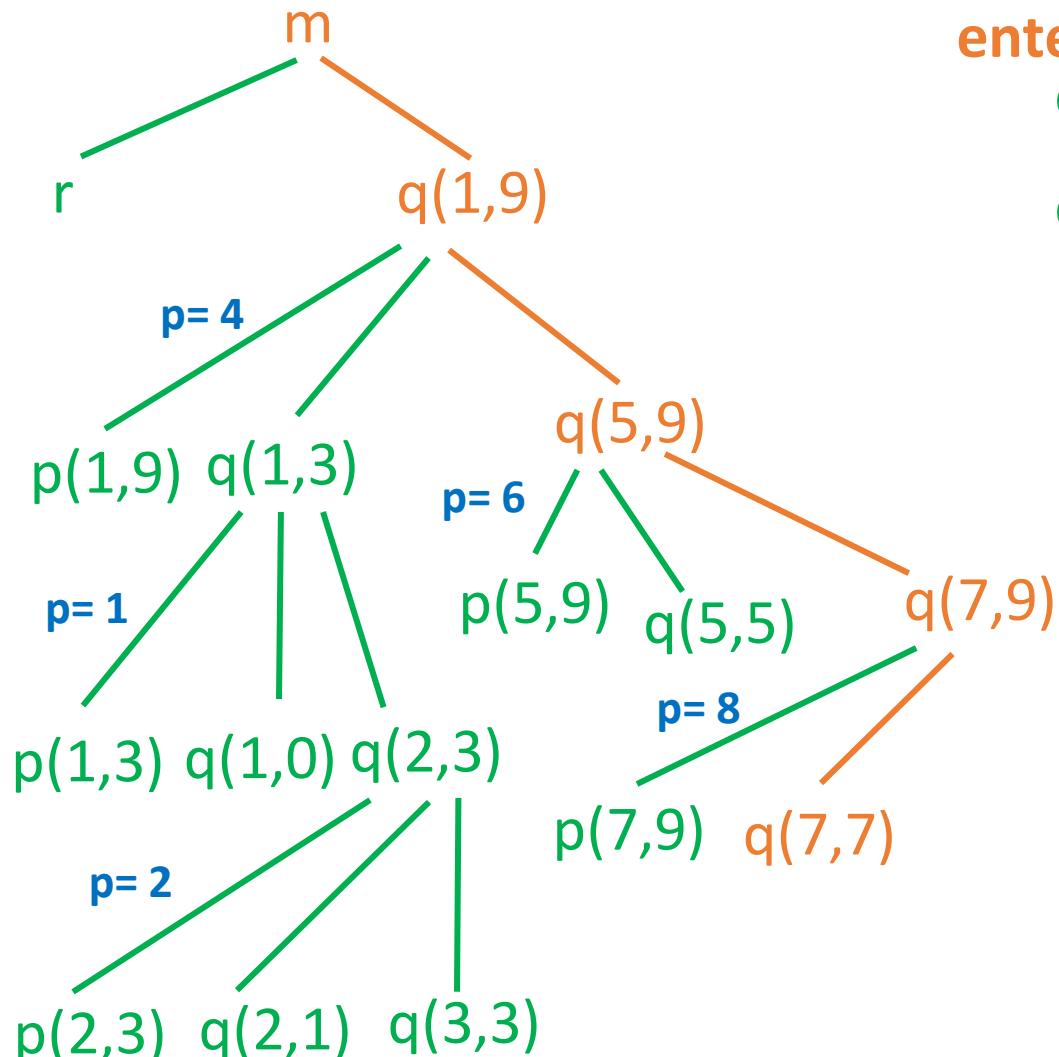
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

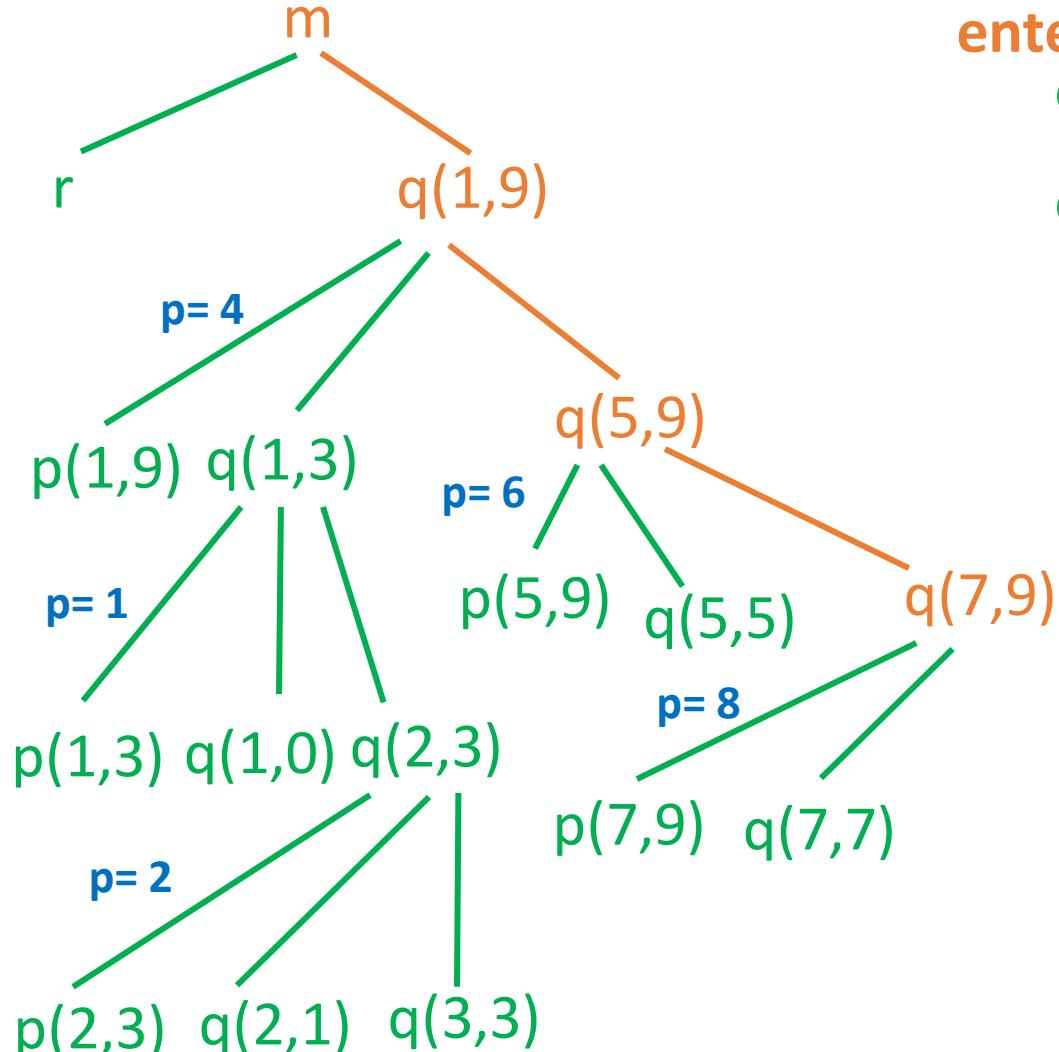
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

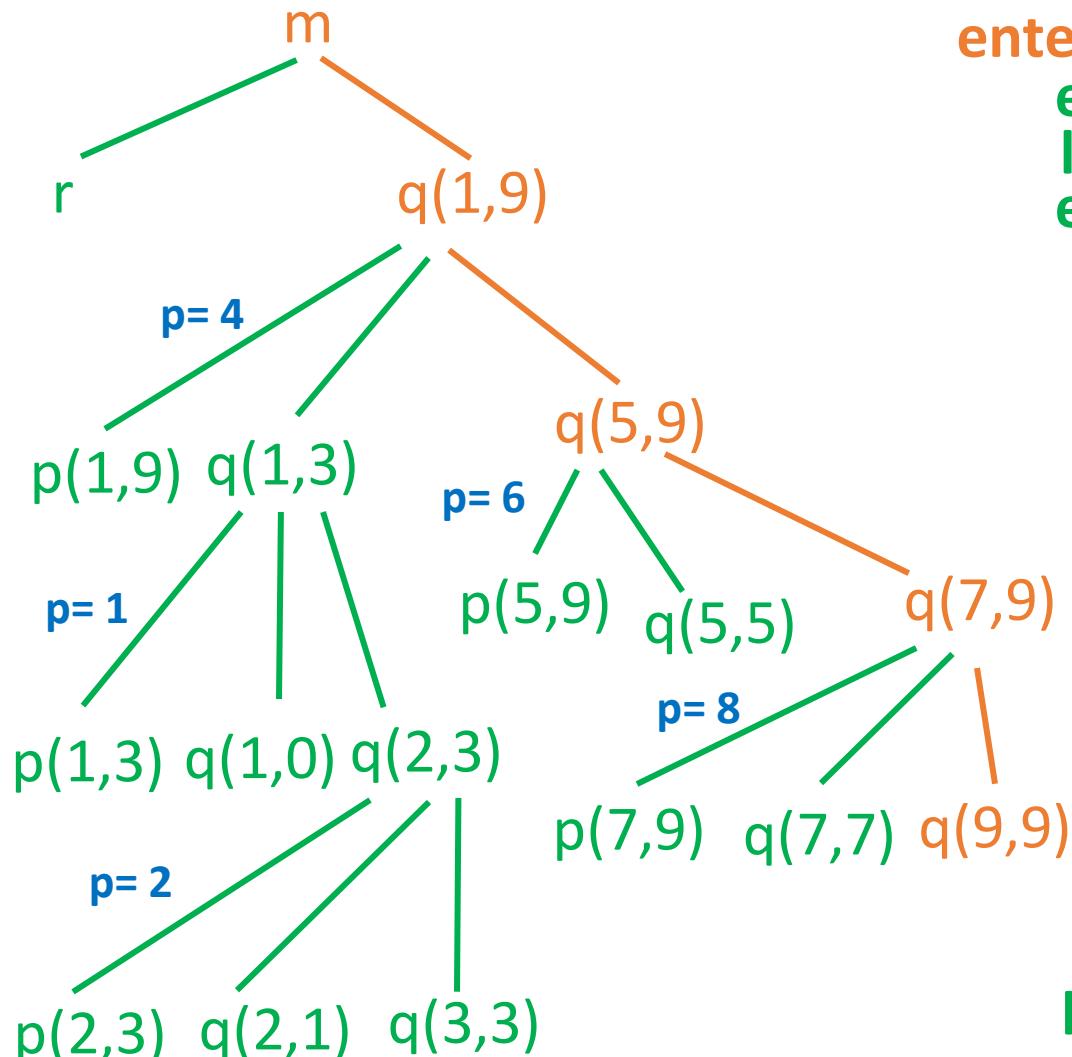


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

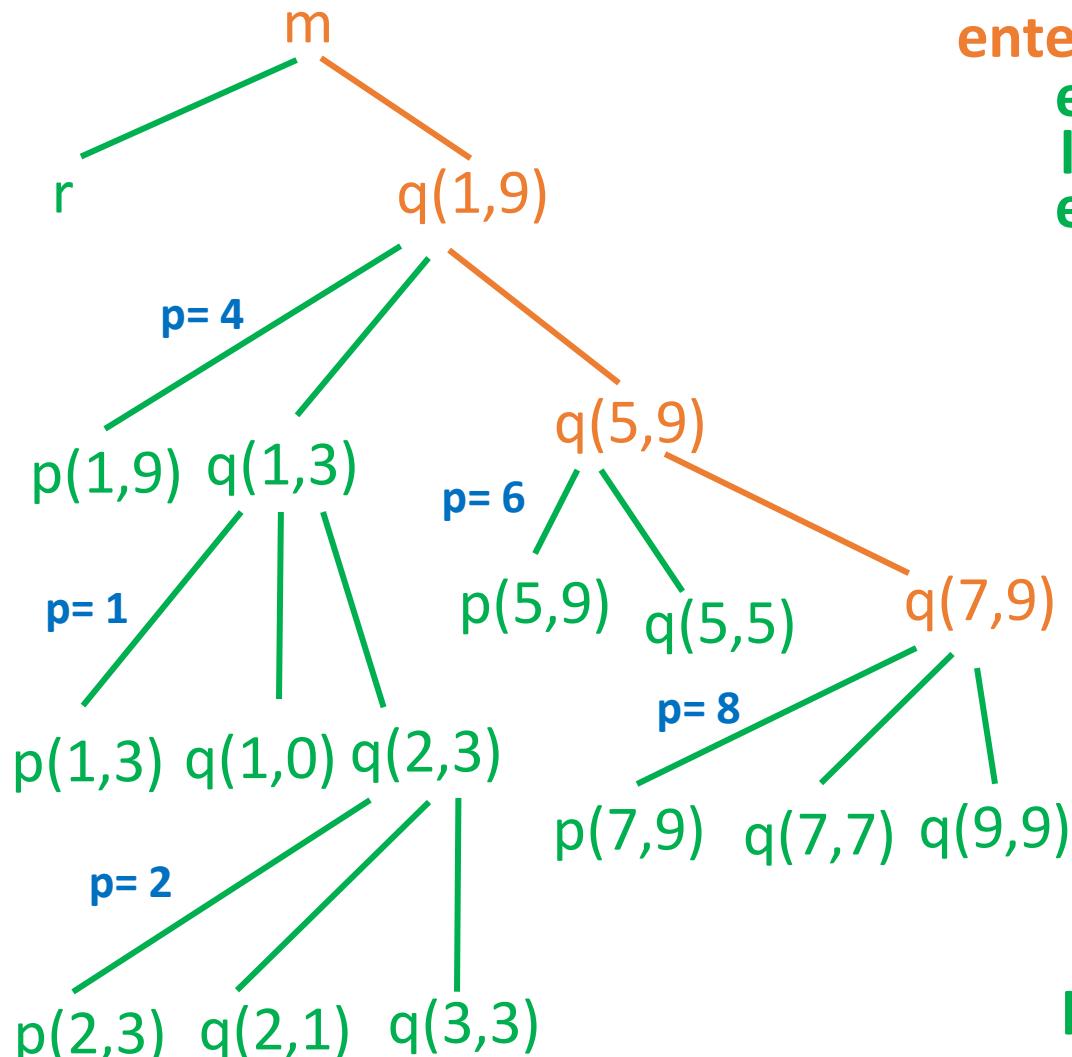
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)

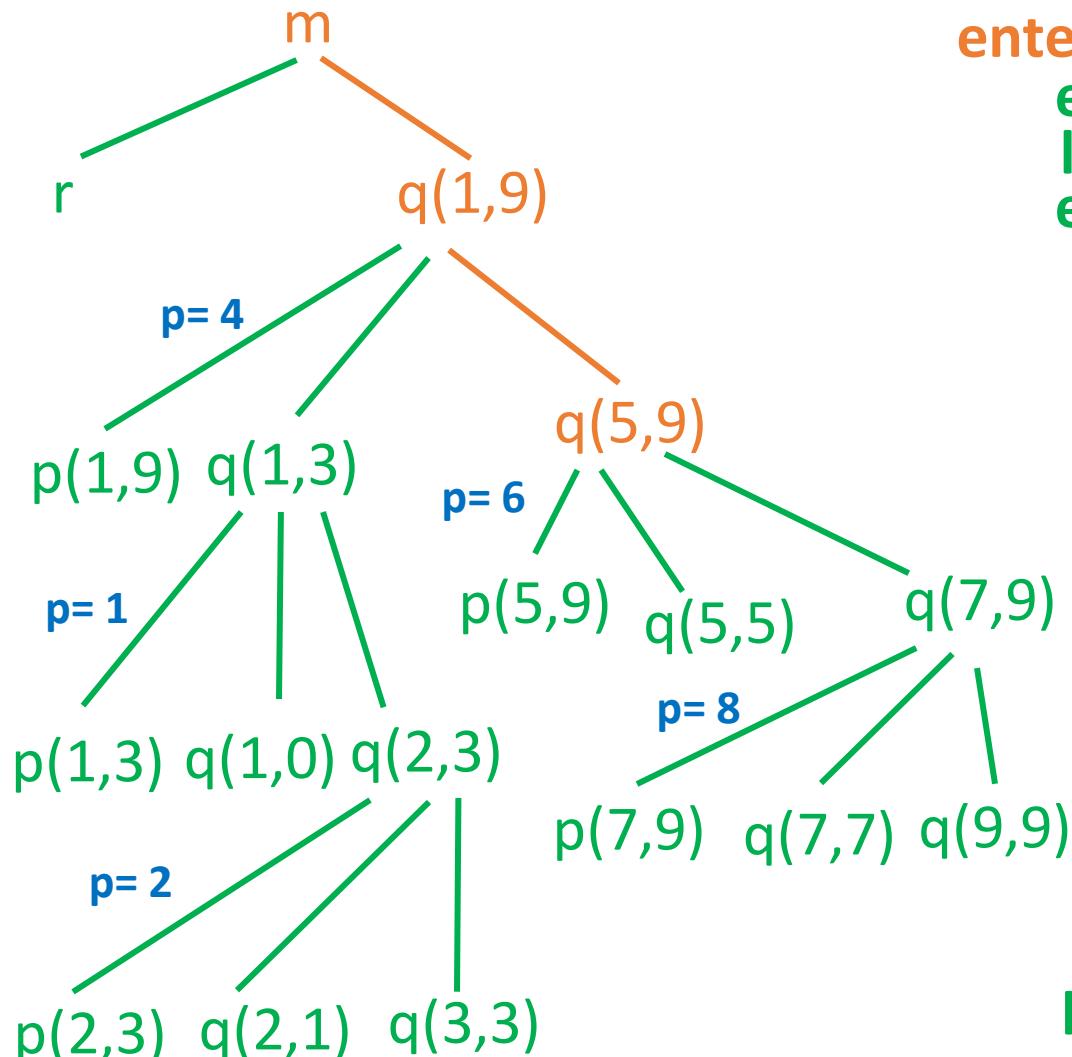


Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) ↑ p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

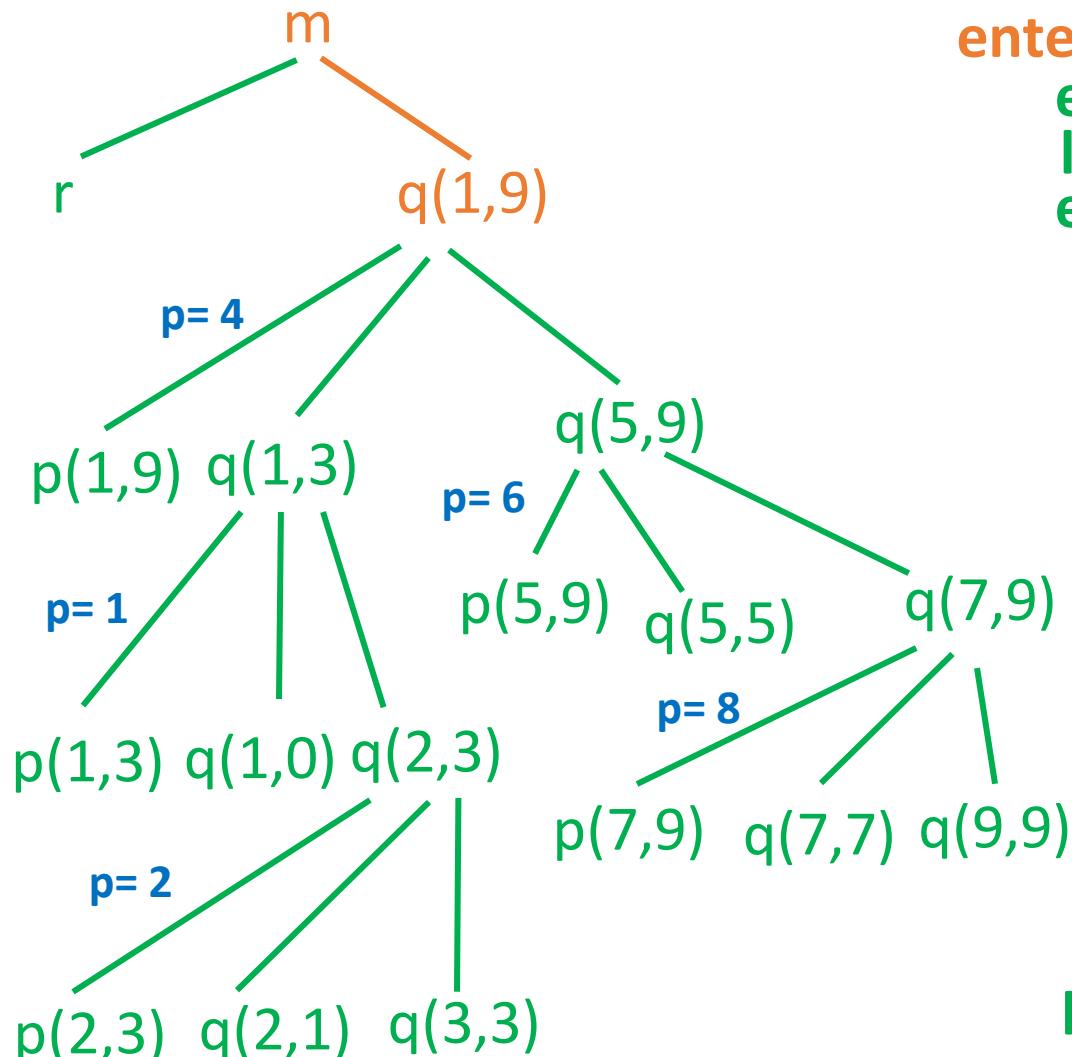
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()  
leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

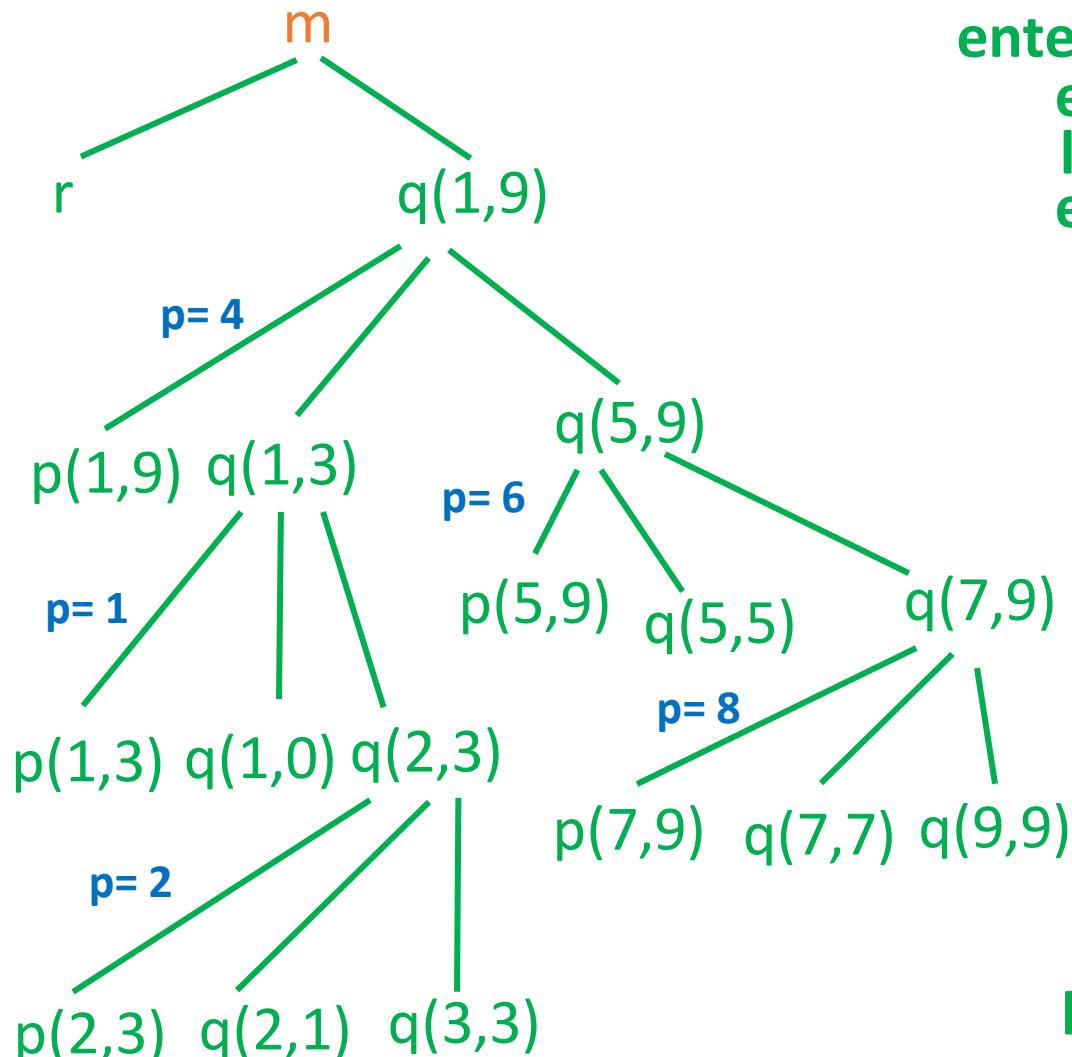
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()

enter readArray ()

leave readArray ()

enter quicksort (1,9)

enter partition (1,9) → p= 4

leave partition (1,9)

enter quicksort (1,3)

enter partition (1,3) → p= 1

leave partition (1,3)

enter quicksort (1,0)

leave quicksort (1,0)

enter quicksort (2,3)

enter partition (2,3) → p= 2

leave partition (2,3)

enter quicksort (2,1)

leave quicksort (2,1)

enter quicksort (3,3)

leave quicksort (3,3)

leave quicksort (2,3)

leave quicksort (1,3)

enter quicksort (5,9)

enter partition (5,9) → p= 6

leave partition (5,9)

enter quicksort (5,5)

leave quicksort (5,5)

enter quicksort (7,9) p= 8

enter partition (7,9)

leave partition (7,9)

enter quicksort (7,7)

leave quicksort (7,7)

enter quicksort (9,9)

leave quicksort (9,9)

leave quicksort (7,9)

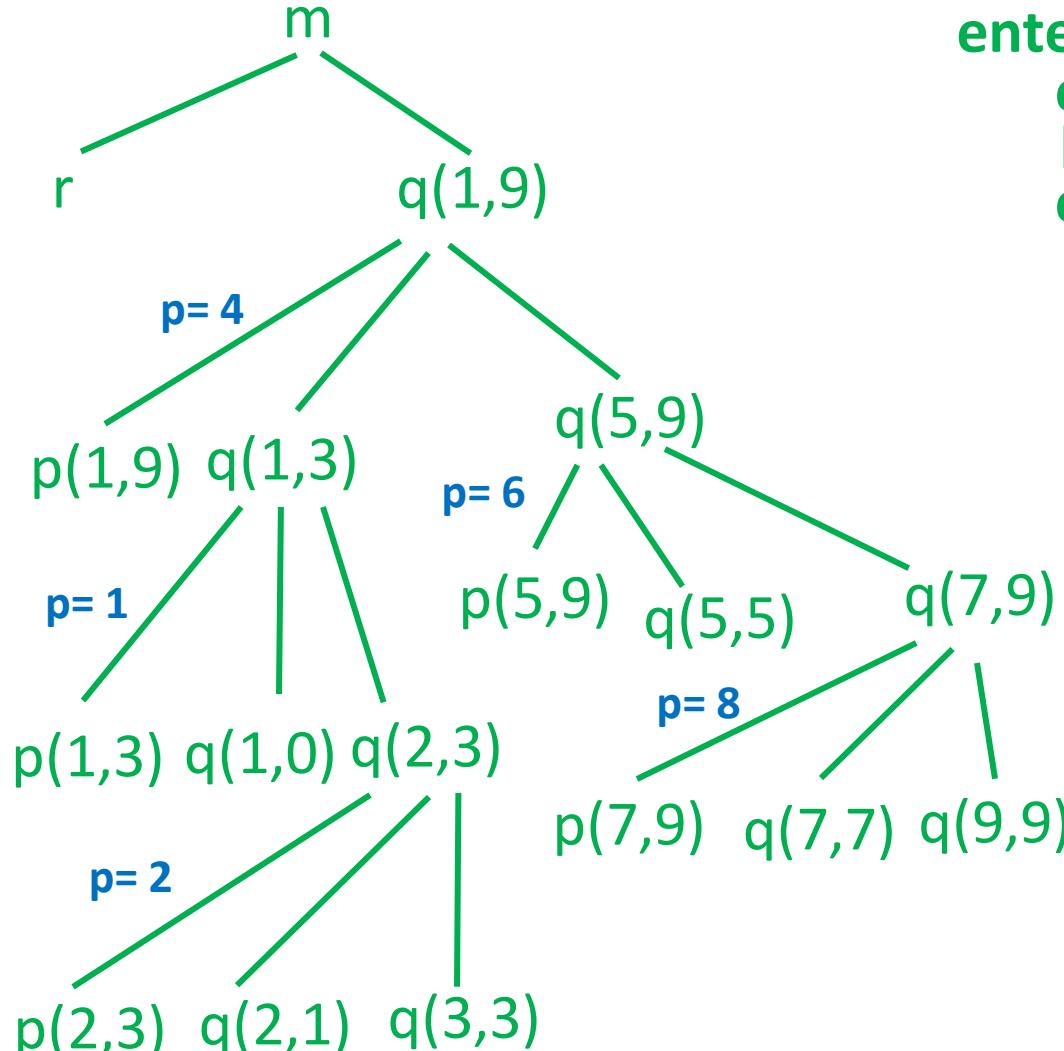
leave quicksort (5,9)

leave quicksort (1,9)

leave main ()

Possible Activations for the Quicksort Program

# Example 7.2(Cont...)



Activation Tree Representing Calls  
During an Execution of Quicksort

enter main ()  
enter readArray ()  
leave readArray ()  
enter quicksort (1,9)  
enter partition (1,9) → p= 4  
leave partition (1,9)  
enter quicksort (1,3)  
enter partition (1,3) → p= 1  
leave partition (1,3)  
enter quicksort (1,0)  
leave quicksort (1,0)  
enter quicksort (2,3)  
enter partition (2,3) → p= 2  
leave partition (2,3)  
enter quicksort (2,1)  
leave quicksort (2,1)  
enter quicksort (3,3)  
leave quicksort (3,3)  
leave quicksort (2,3)  
leave quicksort (1,3)  
enter quicksort (5,9)  
enter partition (5,9) → p= 6  
leave partition (5,9)  
enter quicksort (5,5)  
leave quicksort (5,5)  
enter quicksort (7,9)  
enter partition (7,9) → p= 8  
leave partition (7,9)  
enter quicksort (7,7)  
leave quicksort (7,7)  
enter quicksort (9,9)  
leave quicksort (9,9)  
leave quicksort (7,9)  
leave quicksort (5,9)  
leave quicksort (1,9)  
leave main ()

Possible Activations for the Quicksort Program

## Activation Trees (Cont...)

- **The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:**
  - The sequence of procedure calls corresponds to a preorder traversal of the activation tree
  - The sequence of returns corresponds to a postorder traversal of the activation tree

## Activation Trees (Cont...)

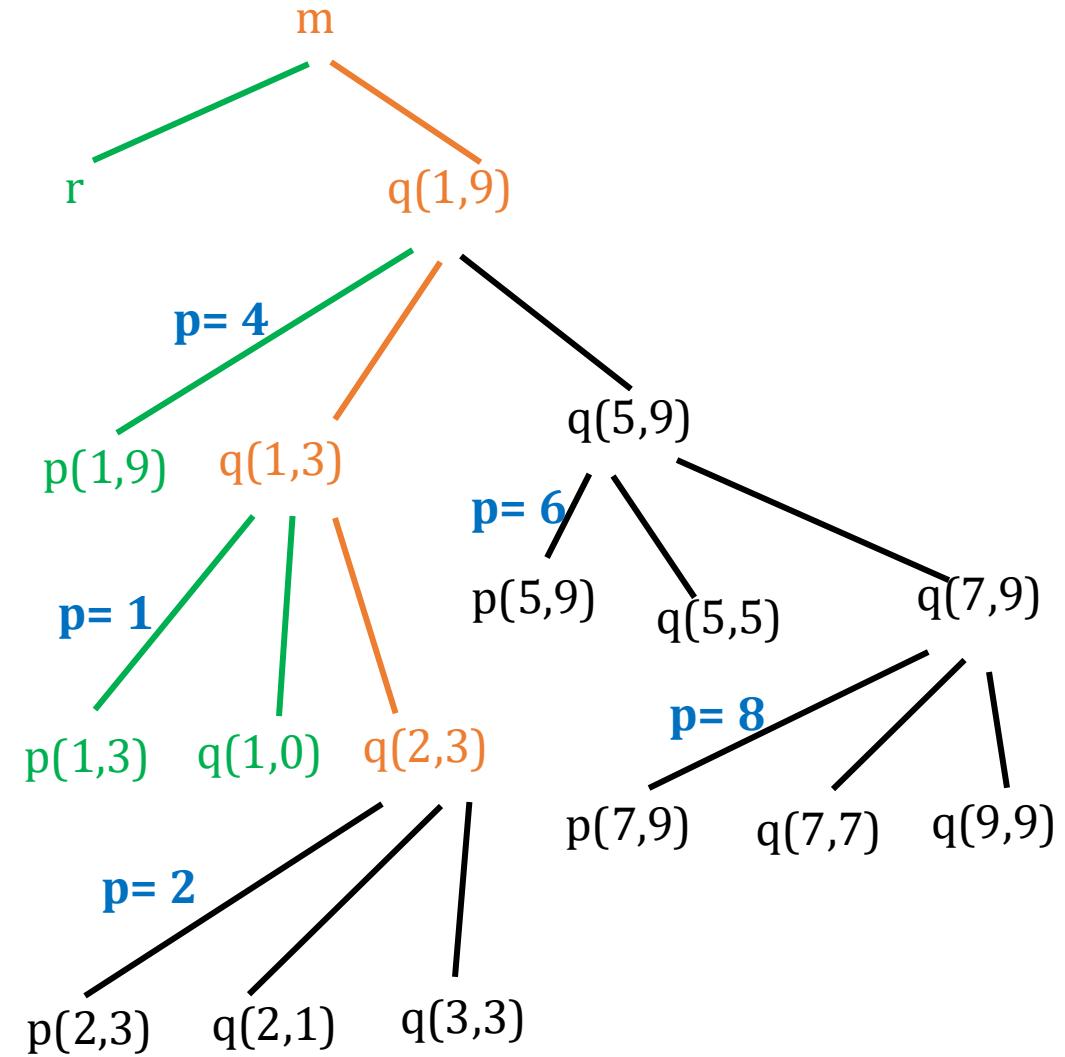
- **The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:**
- Suppose that control lies within a particular activation of some procedure corresponding to a node N of the activation tree
- Then the activations that are currently open (live) are those that correspond to node N and its ancestors
- The order in which these activations were called is the order in which they appear along the path to N starting at the root and they will return in the reverse of that order

# Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack
- Each live activation has an activation record(sometimes called a frame) on the control stack
  - With the root of the activation tree at the bottom and
  - The entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides
- The latter activation has its record at the top of the stack

## Example 7.3

- If control is currently in the activation  $q(2,3)$  of the tree, then the activation record for  $q(2,3)$  is at the top of the control stack
- Just below is the activation record for  $q(1,3)$ , the parent of  $q(2,3)$  in the tree
- Below that is the activation record for  $q(1,9)$ , and
- At the bottom is the activation record for  $m$ , the main function and root of the activation tree



**Activation Tree Representing Calls  
During an Execution of Quicksort**

# Contents of an Activation Record

- The contents of activation records vary with the language being implemented
- Here is a list of the kinds of data that might appear in an activation record
- See figure for a summary and possible order for these elements

<b>Actual Parameters</b>
<b>Returned Values</b>
<b>Control Link</b>
<b>Access Link</b>
<b>Saved Machine Status</b>
<b>Local Data</b>
<b>Temporaries</b>

**A General Activation Record**

# Contents of an Activation Record (Cont...)

- **Temporary values** such as those arising from the evaluation of expressions in cases where those temporaries cannot be held in registers
- For example, compiler generated temporaries like  $t_1, t_2$  used to hold temporary values will be stored in **temporary** section

<b>Actual Parameters</b>
<b>Returned Values</b>
<b>Control Link</b>
<b>Access Link</b>
<b>Saved Machine Status</b>
<b>Local Data</b>
<b>Temporaries</b>

**A General Activation Record**

# Contents of an Activation Record (Cont...)

- **Local data** belonging to the procedure whose activation record this is

- For example,

```
int sumfunc (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

- **sum** is a local variable of the **sumfunc** function and it will be stored in **local data** section

Actual Parameters
Returned Values
Control Link
Access Link
Saved Machine Status
<b>Local Data</b>
Temporaries

A General Activation Record

# Contents of an Activation Record (Cont...)

- A **saved machine status** with information about the state of the machine just before the call to the procedure
- This information typically includes the return address (For example, value of the program counter to which the called procedure must return) and
- The contents of registers that were used by the calling procedure and that must be restored when the return occurs

Actual Parameters
Returned Values
Control Link
Access Link
<b>Saved Machine Status</b>
Local Data
Temporaries

A General Activation Record

# Contents of an Activation Record (Cont...)

- An **access link** which is a pointer may be needed to locate data needed by the called procedure but found elsewhere e.g. in another activation record

Actual Parameters
Returned Values
Control Link
<b>Access Link</b>
Saved Machine Status
Local Data
Temporaries

A General Activation Record

## Contents of an Activation Record (Cont...)

- For example, procedure **p()** needs data which is stored in the activation record of procedure **q()**
- Then the **access link** of **p()** will point to the **access link** of **q()**

Actual Parameters
Returned Values
Control Link
<b>Access Link</b>
Saved Machine Status
Local Data
Temporaries

Activation Record of p()

Actual Parameters
Returned Values
Control Link
<b>Access Link</b>
Saved Machine Status
Local Data
Temporaries

Activation Record of q()



# Contents of an Activation Record (Cont...)

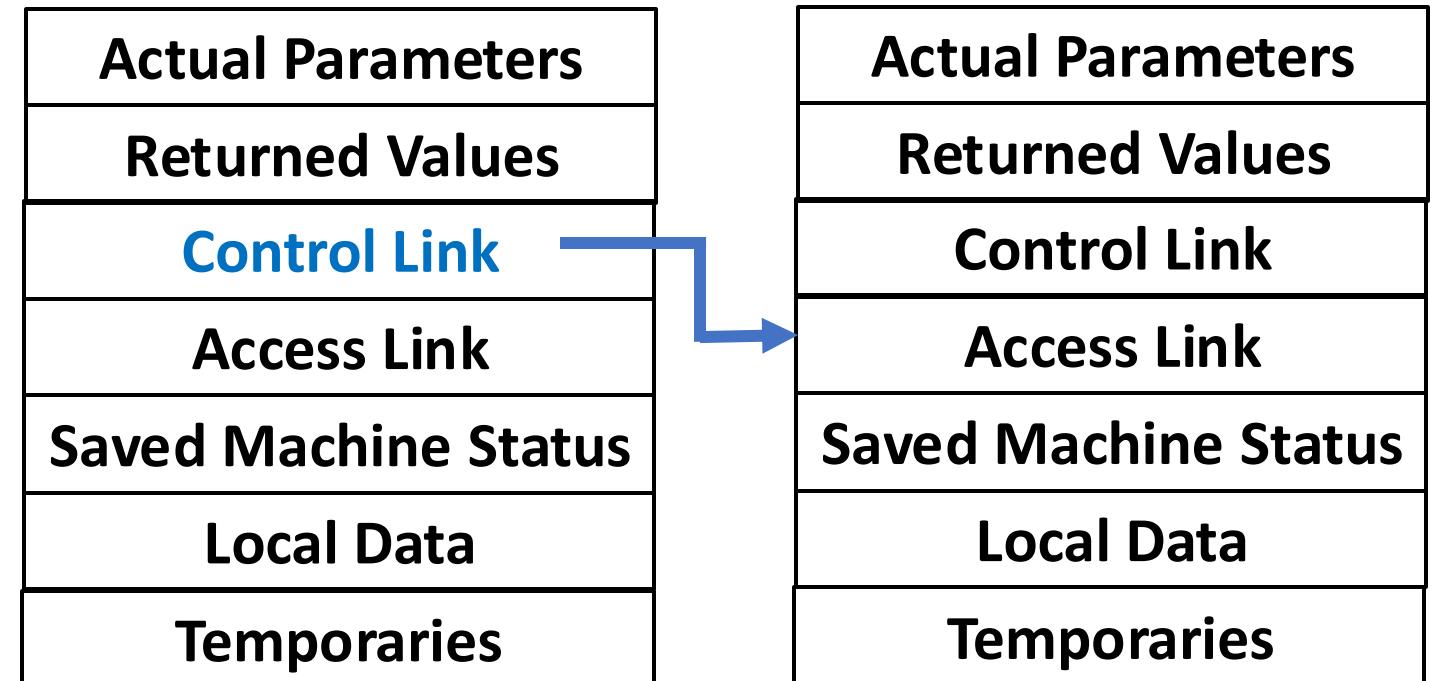
- A **control link** which is another pointer pointing to the activation record of the caller

Actual Parameters
Returned Values
<b>Control Link</b>
Access Link
Saved Machine Status
Local Data
Temporaries

A General Activation Record

# Contents of an Activation Record (Cont...)

- For example, procedure **q()** calls procedure **p()**
- Then the **control link** of **p()** will point to the activation record of **q()**



# Contents of an Activation Record(Cont...)

- Space for the **return value** of the called function, if any as not all called procedures return a value
- For example,

```
int sumfunc (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

- The **sumfunc** function returns value of **sum** and it will be stored in **return value** section but we may prefer to place that value in a register for efficiency

Actual Parameters
Returned Values
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

A General Activation Record

# Contents of an Activation Record (Cont...)

- The **actual parameters** used by the calling procedure
- For example,

```
int sumfunc (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

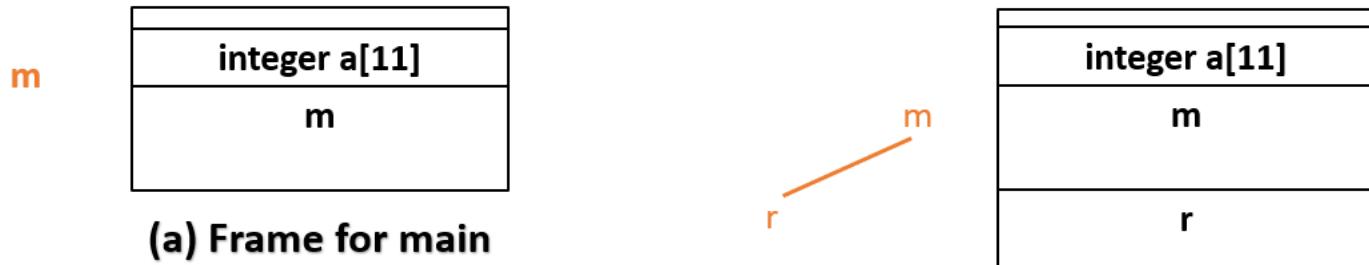
- **a** and **b** are the actual parameters of the **sumfunc** function and it will be stored in **actual parameters** section but we may prefer to place that value in a register for efficiency

Actual Parameters
Returned Values
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

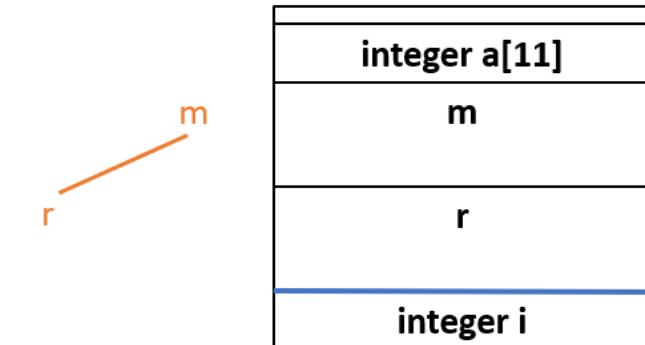
A General Activation Record

## Example 7.4

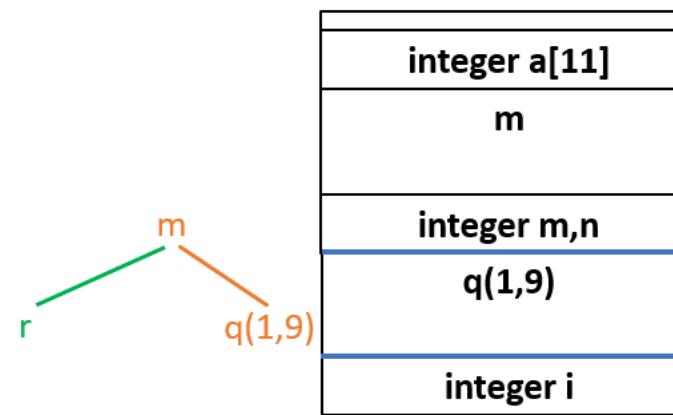
- Figure shows several snapshots of the runtime stack as control flows through the activation tree
- Green lines in the partial trees go to activations that have ended and orange lines indicates they are currently activated



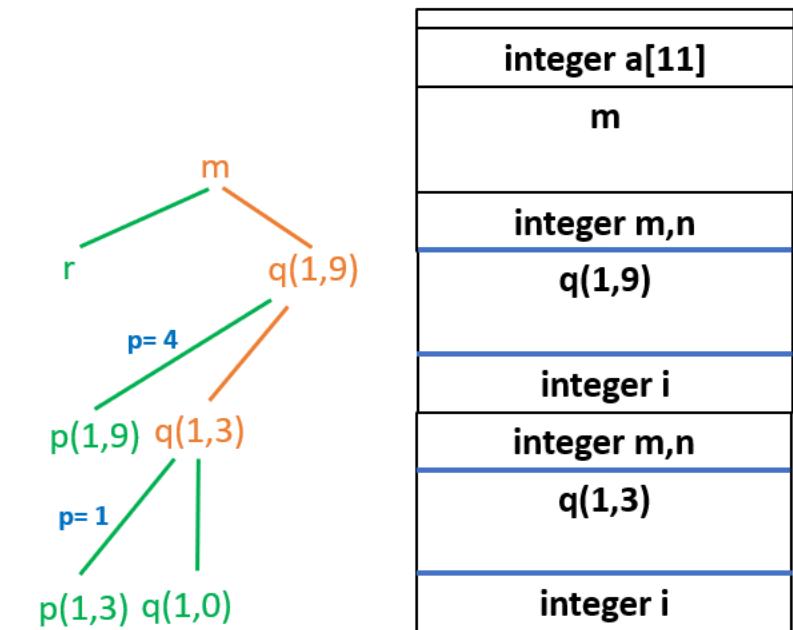
(a) Frame for main



(b) r is activated



(c) r has been popped  
and q(1,9) pushed



(d) Control returns to  
q(1,3)

Downward-growing stack of Activation Records

## Example 7.4 (Cont...)

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}

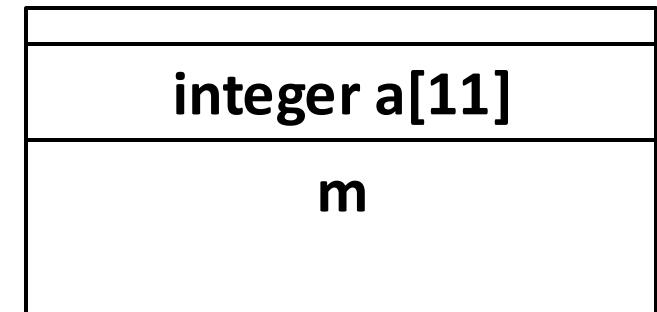
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m .. n] so that
       a[m .. p - 1] are less than v, a[p] = v, and a[p + 1 .. n] are
       equal to or greater than v. Returns p. */
    ...
}

void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

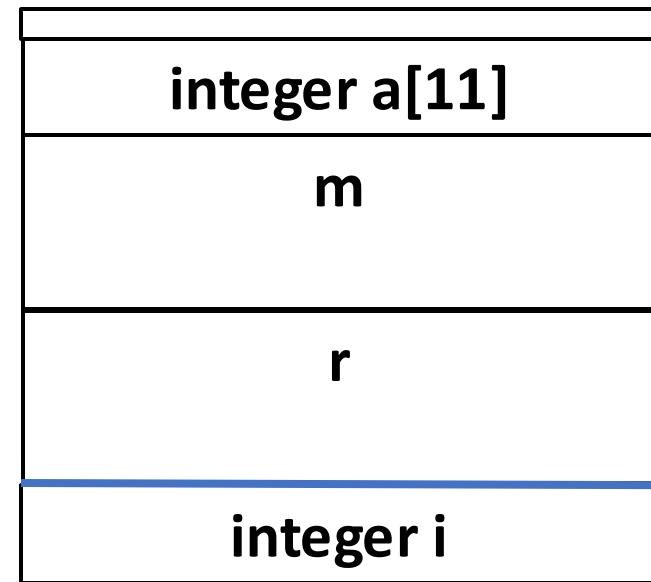
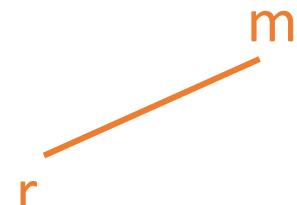
Since array a is global, space is allocated for it before execution begins with an activation of procedure main as shown in (a)



(a) Frame for main

## Example 7.4 (Cont...)

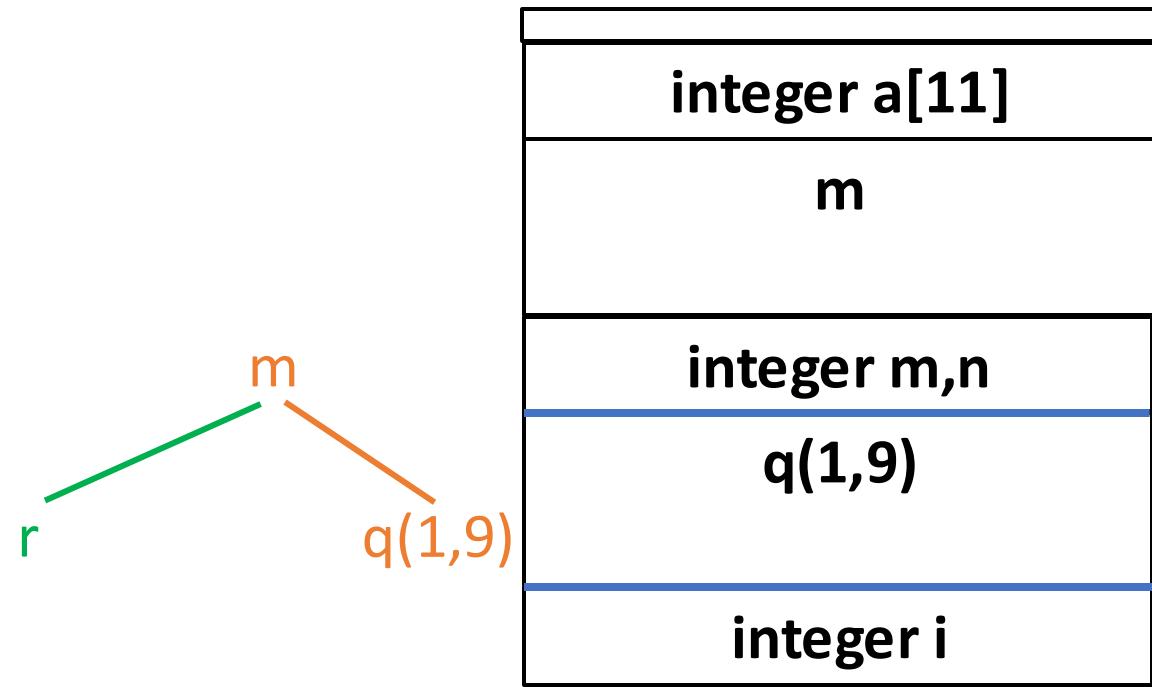
- When control reaches the first call in the body of main procedure r is activated and its activation record is pushed onto the stack (b)
- The activation record for r contains space for local variable i
- The top of stack is at the bottom of diagrams



**(b) r is activated**

## Example 7.4 (Cont...)

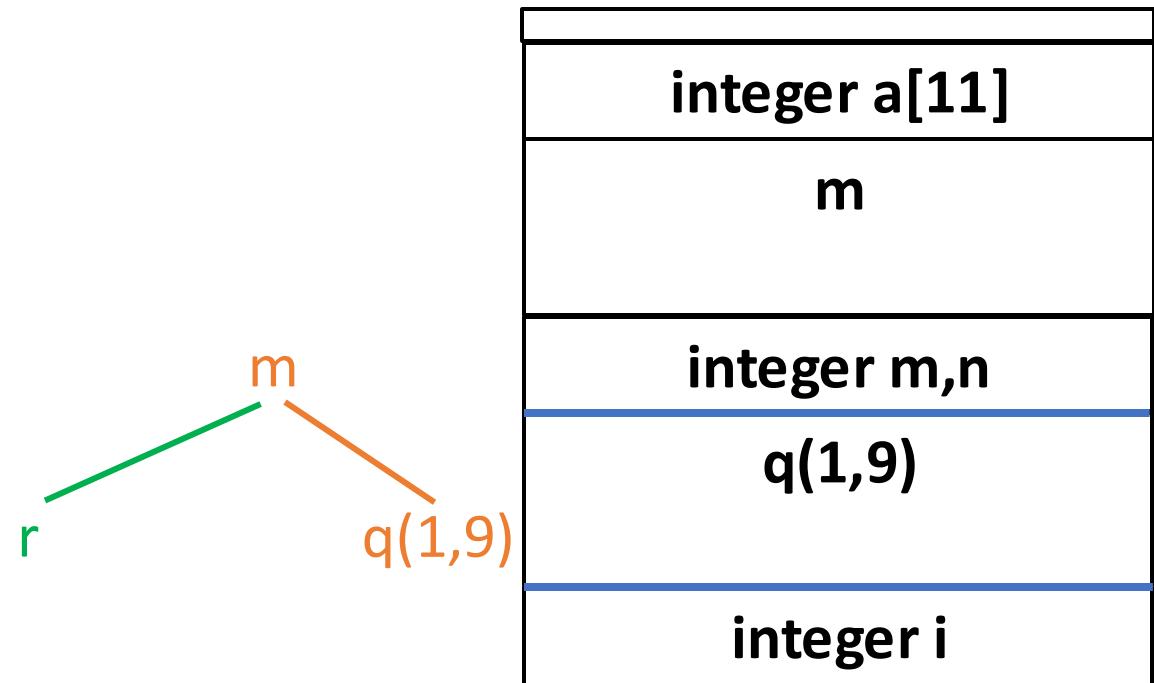
- When control returns from this activation its record is popped leaving just the record for main on the stack
- Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack as in (c)
- The activation record for q contains space for the parameters m and n and the local variable i following the general layout



**(c) r has been popped  
and q(1,9) pushed**

## Example 7.4 (Cont...)

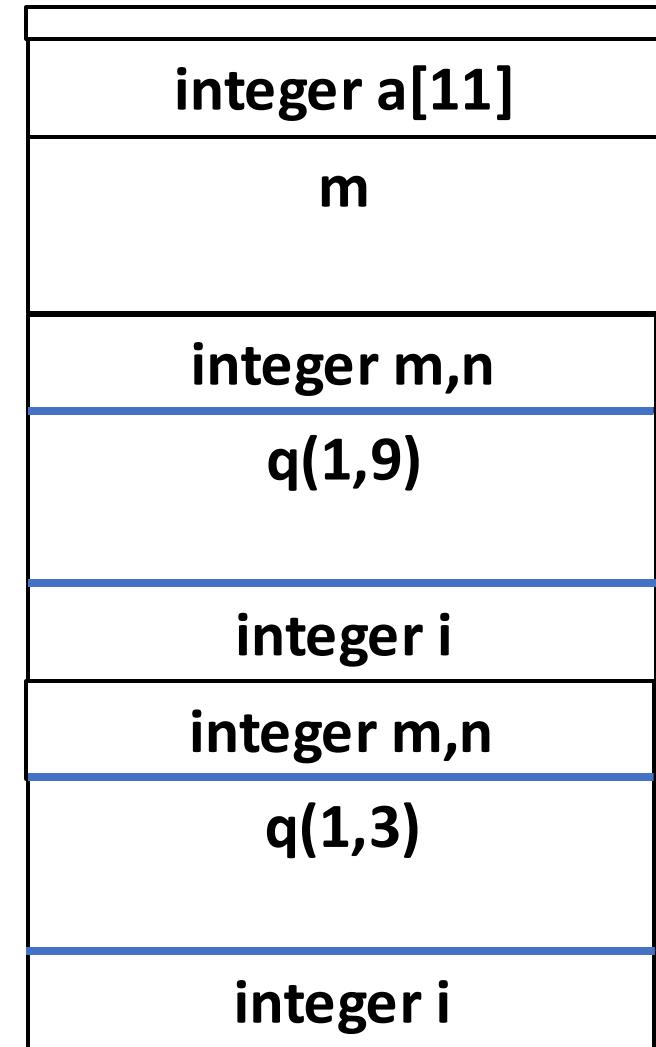
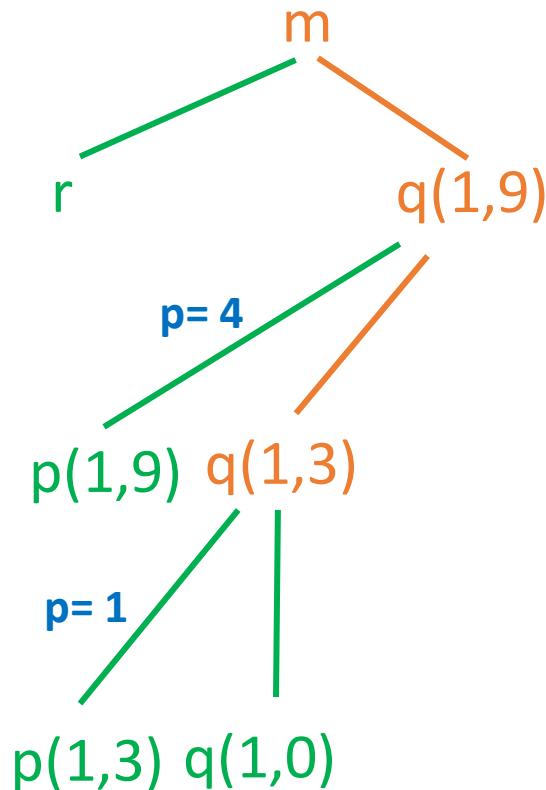
- Notice that space once used by the call of r is reused on the stack
- No trace of data local to r will be available to q(1,9)
- When q(1,9) will return then the stack again will hold only the activation record for main



**(c) r has been popped  
and q(1,9) pushed**

## Example 7.4 (Cont...)

- Several activations occur between the last two snapshots
- A recursive call to  $q(1,3)$  was made
- Activations  $p(1,3)$  and  $q(1,0)$  have begun and ended during the lifetime of  $q(1,3)$  leaving the activation record for  $q(1,3)$  on top of (d)
- Notice that when a procedure is recursive it is normal to have several of its activation records on the stack at the same time



(d) Control returns to  
 $q(1,3)$

# Calling Sequences

1. Values communicated between caller and callee are generally placed at the beginning of the callee's activation record.
2. Fixed-length items are generally placed in the middle which includes the control link, the access link, and the machine status fields.
3. Items whose size may not be known early enough are placed at the end of the activation record
4. Must locate the top-of-stack pointer to have it point to the end of the fixed-length fields in the activation record.

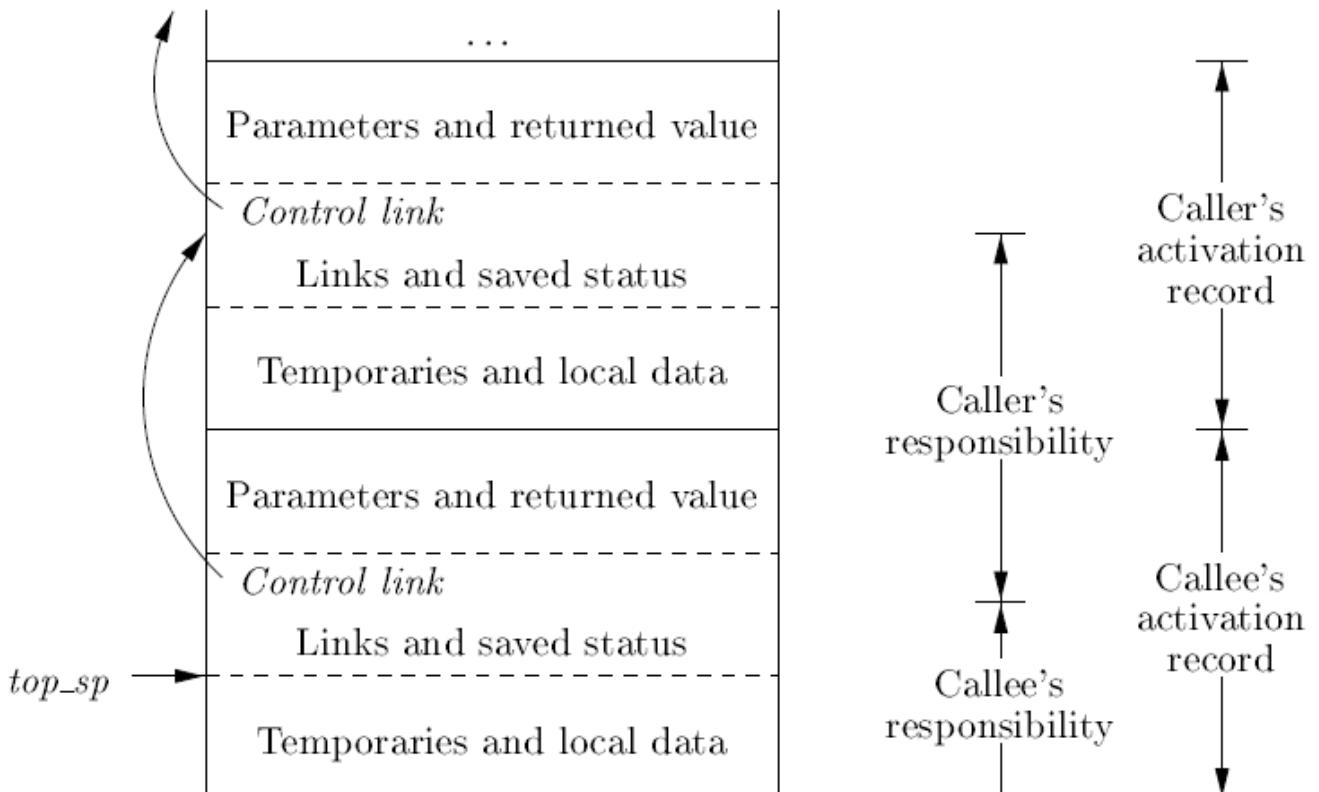


Figure 7.7: Division of tasks between caller and callee

# Variable-Length Data on the Stack

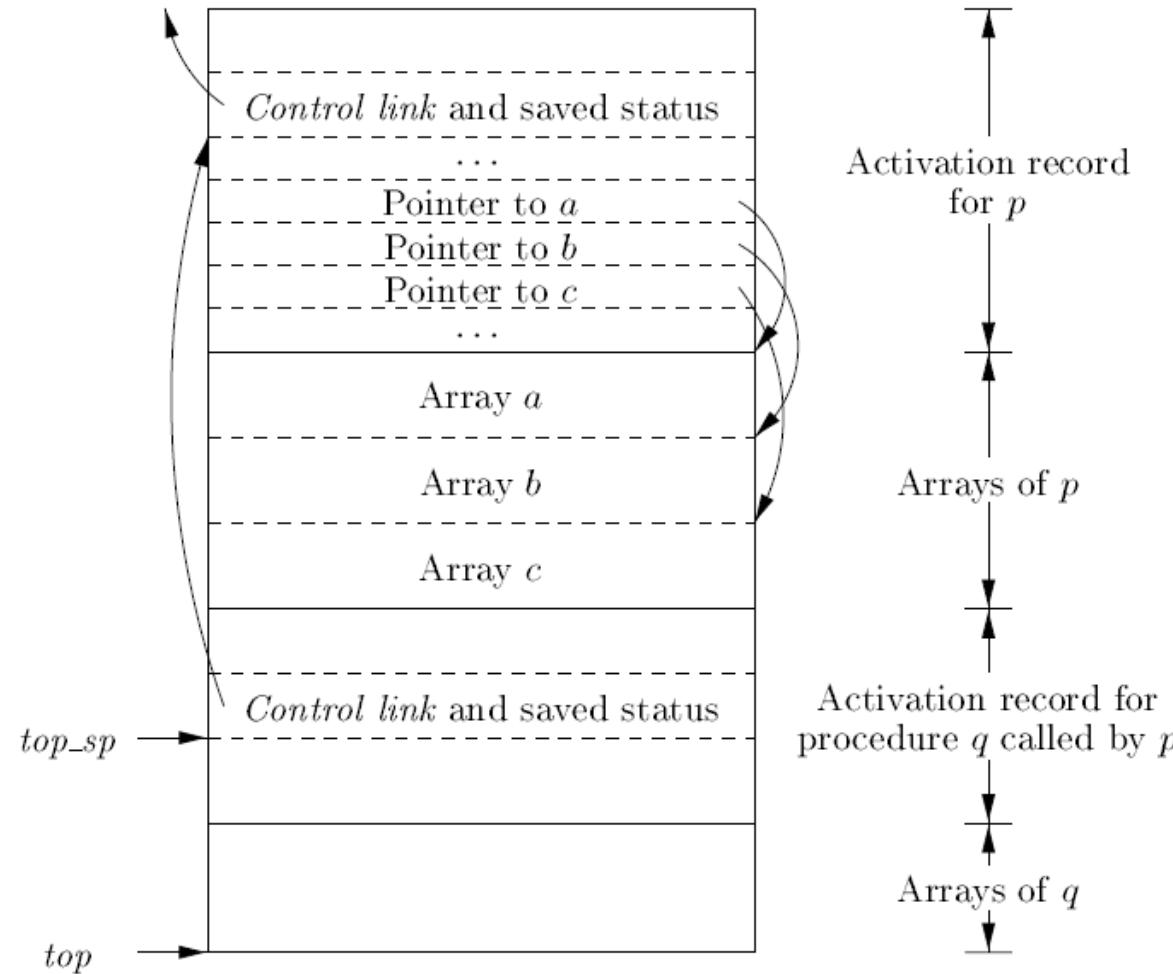


Figure 7.8: Access to dynamically allocated arrays

# Access to Nonlocal Data on the Stack

- Data Access Without Nested Procedures
- Issues With Nested Procedures
- Nesting Depth
- Access Link
- Access Link for procedure parameter

# Quicksort using Nested functions

```
1) fun sort(inputFile, outputFile) =  
    let  
2)        val a = array(11,0);  
3)        fun readArray(inputFile) = ...  
            ... a ... ;  
4)        fun exchange(i,j) =  
            ... a ... ;  
5)        fun quicksort(m,n) =  
            let  
6)                val v = ... ;  
7)                fun partition(y,z) =  
                    ... a ... v ... exchange ...  
8)            in  
9)                ... a ... v ... partition ... quicksort  
10)           end  
11)          in  
12)              ... a ... readArray ... quicksort ...  
end;
```

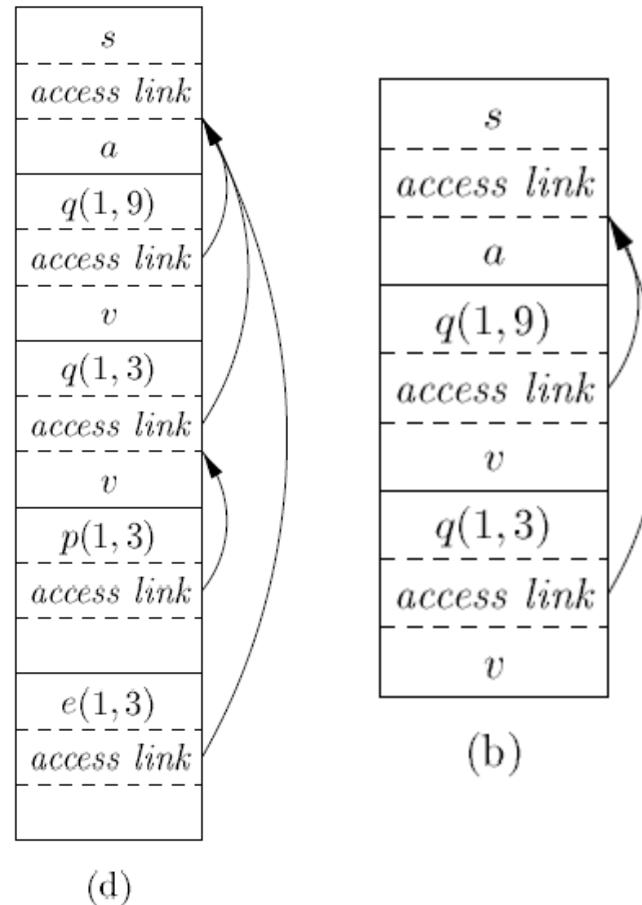


Figure 7.10: A version of quicksort, in ML style, using nested functions

# Quicksort using Nested functions

```
1) fun sort(inputFile, outputFile) =  
    let  
2)      val a = array(11,0);  
3)      fun readArray(inputFile) = ...  
4)          ... a ... ;  
5)      fun exchange(i,j) =  
          ... a ... ;  
6)      fun quicksort(m,n) =  
          let  
7)              val v = ... ;  
8)              fun partition(y,z) =  
                  ... a ... v ... exchange ...  
9)              in  
10)                 ... a ... v ... partition ... quicksort  
11)             end  
12)         in  
13)             ... a ... readArray ... quicksort ...  
14)         end;
```

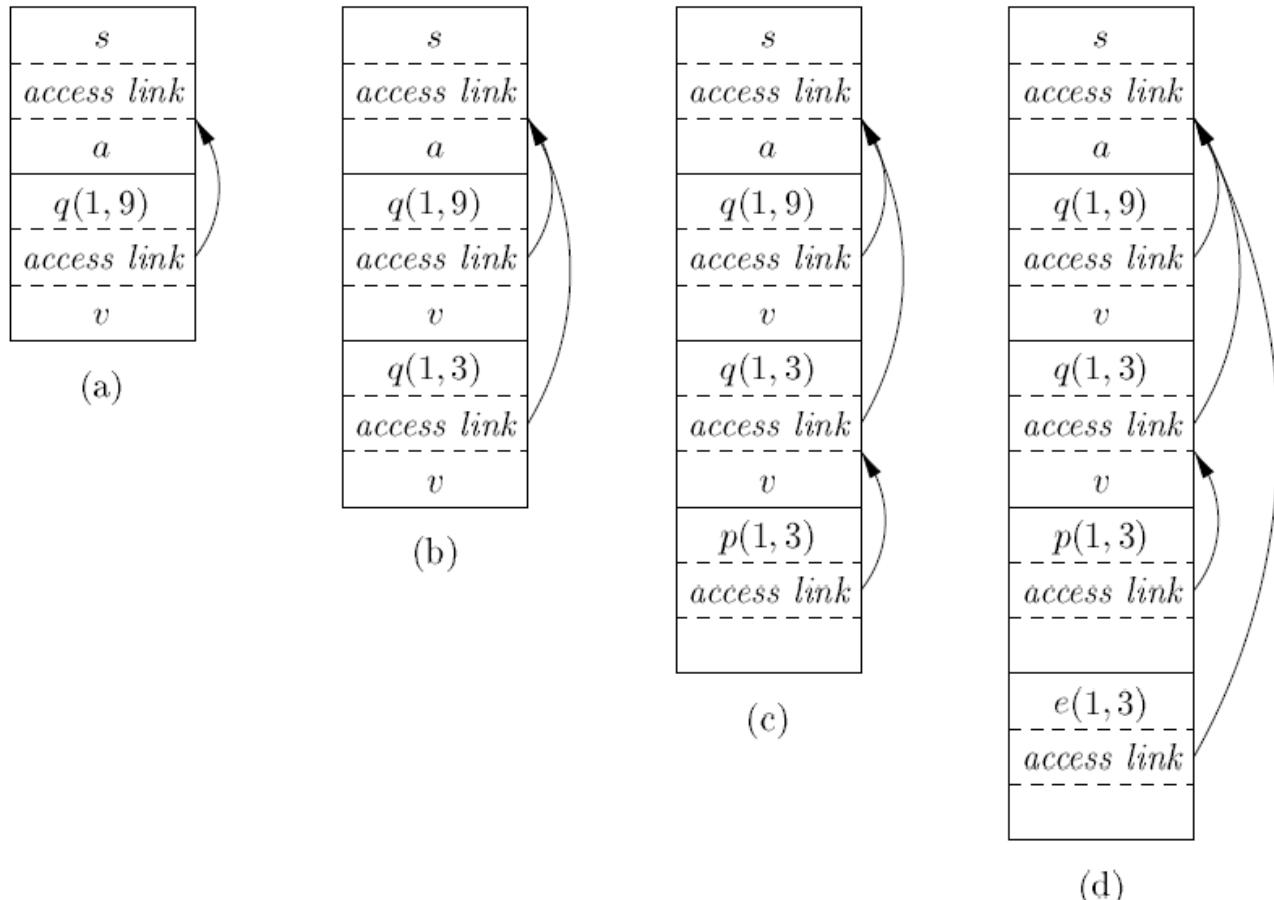
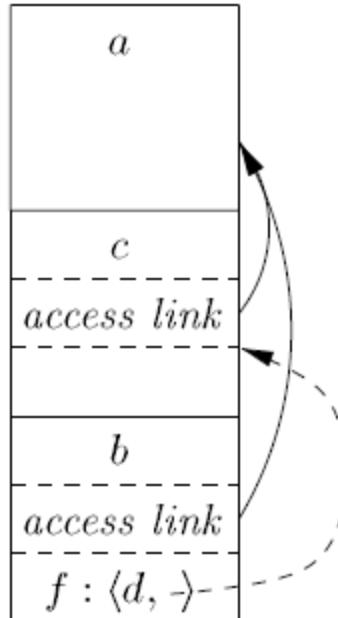


Figure 7.10: A version of quicksort, in ML style, using nested functions

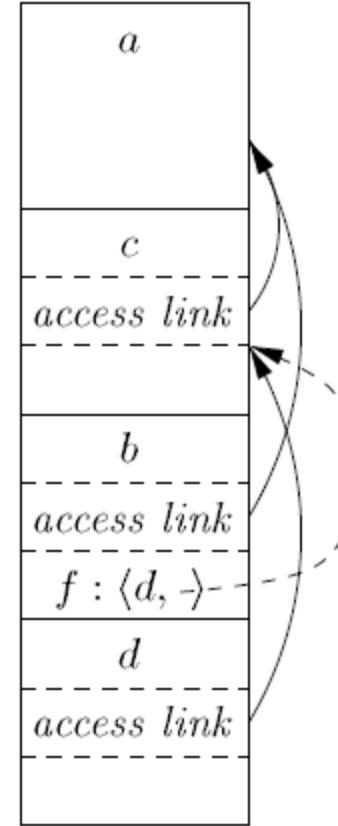
Figure 7.11: Access links for finding nonlocal data

# Access Links for Procedure Parameters

```
fun a(x) =  
  let  
    fun b(f) =  
      ... f ... ;  
    fun c(y) =  
      let  
        fun d(z) = ...  
        in  
          ... b(d) ...  
      end  
    in  
      ... c(1) ...  
  end;
```



(a)



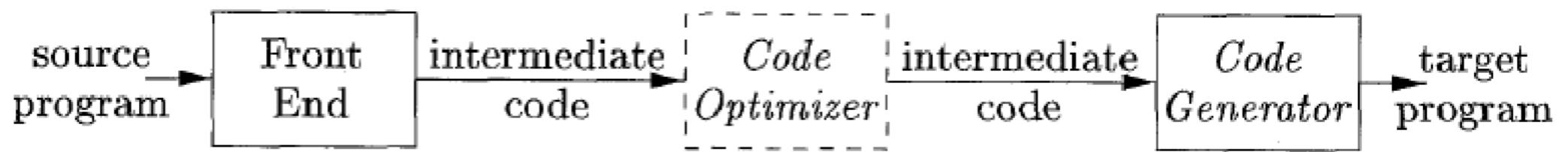
(b)

**END**

# **Chapter-08**

# **Code Generation**

# Code Generation



Position of code generator

# Issues in the Design of a Code Generator

While the details are dependent on the specifics of

- the intermediate representation,
- the target language and
- the run-time system

tasks such as

- instruction selection,
- register allocation and assignment and
- instruction ordering

are encountered in the design of almost all code generators

# **Issues in the Design of a Code Generator (Cont...)**

- The most important criterion for a code generator is that it produces correct code.
- Correctness takes on special significance because of the number of special cases that a code generator might face.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested and maintained is an important design goal.

# Input to the Code Generator

- The input to the code generator is
  - The **intermediate representation (IR)** of the source program produced by the front end
  - Along with information in the **symbol table** that is used to determine the run-time addresses of the data objects denoted by the names in the IR
- The many choices for the IR include
  - Three-address representations such as quadruples, triples, indirect triples
  - Virtual machine representations such as byte codes and stack-machine code
  - Linear representations such as postfix notation and
  - Graphical representations such as syntax trees and DAG's

# Input to the Code Generator (Cont...)

- We assume that the front end has
    - scanned,
    - parsed and
    - translated the source program into a relatively low-level IR
- so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate such as integers and floating-point numbers

# **Input to the Code Generator (Cont...)**

- We also assume that
  - all syntactic and static semantic errors have been detected
  - the necessary type checking has taken place and
  - type conversion operators have been inserted wherever necessary
- The code generator can therefore proceed on the assumption that its input is free of these kinds of errors

# The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code
- The most common target-machine architectures are
  - RISC (reduced instruction set computer),
  - CISC (complex instruction set computer), and
  - stack based

# The Target Program (Cont...)

- A RISC machine typically has
  - ✓ Many registers
  - ✓ Three-address instructions
  - ✓ Simple addressing modes and
  - ✓ Relatively simple instruction-set architecture

# The Target Program (Cont...)

- In contrast, a CISC machine typically has
  - ✓ Few registers,
  - ✓ Two-address instructions,
  - ✓ A variety of addressing modes,
  - ✓ Several register classes,
  - ✓ Variable-length instructions and
  - ✓ Instructions with side effects

## The Target Program (Cont...)

- In a stack-based machine operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack
- To achieve high performance the top of the stack is typically kept in registers
- Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many copy and swap operations.

# The Target Program (Cont...)

- We shall use a very simple RISC-like computer as our target machine
- We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines
- For readability, we use assembly code as the target language
- As long as addresses can be calculated from offsets and other information stored in the symbol table the code generator can produce absolute addresses for names

# Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine
- The complexity of performing this mapping is determined by factors such as
  - the level of the IR,
  - the nature of the instruction-set architecture,
  - the desired quality of the generated code

# Instruction Selection (Cont...)

- If the IR is high level the code generator may translate each IR statement into a sequence of machine instructions using code templates
- Such statement-by-statement code generation, however, often produces poor code that needs further optimization
- If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences

# Instruction Selection (Cont...)

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection
- For example, the uniformity and completeness of the instruction set are important factors
- If the target machine does not support each data type in a uniform manner then each exception to the general rule requires special handling
- On some machines, for example, floating-point operations are done using separate registers

# Instruction Selection (Cont...)

- Instruction speed is another important factor
- If we do not care about the efficiency of the target program then instruction selection is straightforward
- For each type of three-address statement we can design a code skeleton that defines the target code to be generated for that construct

# Instruction Selection (Cont...)

- For example, every three-address statement of the form **x = y + z**, where **x**, **y** and **z** are statically allocated, can be translated into the code sequence

**LD R0, y** // R0 = y (load y into register R0)

**LD R1, z** // R1 = z (load z into register R1)

**ADD R0, R0, R1** // R0 = R0 + R1 (add R1 to R0)

**ST x, R0** // x = R0 (store R0 into x)

# Instruction Selection (Cont...)

- This strategy often produces redundant loads and stores
- For example, the sequence of three-address statements

a = b + c  
d = a + e

would be translated into

Here, the fifth statement is redundant since it loads a value that has just been stored.

LD R0, b	// R0 = b
LD R1, c	// R1 = c
ADD R0, R0, R1	// R0 = R0 + R1
ST a, R0	// a = R0
LD R0, a	// R0 = a
LD R1, e	// R1 = e
ADD R0, R0, R1	// R0 = R0 + R1
ST d, R0	// d = R0

# Instruction Selection (Cont...)

- The quality of the generated code is usually determined by its **speed** and **size**
- On most machines, a given IR program can be implemented by many different code sequences with significant cost differences between the different implementations
- A **naive translation** of the intermediate code may therefore lead to **correct** but **unacceptably inefficient** target code
- We need to know instruction costs in order to design good code sequences but unfortunately accurate cost information is often difficult to obtain

# Instruction Selection (Cont...)

- For example, if the target machine has an “increment” instruction (INC) then the three-address statement **a = a + 1** may be implemented more efficiently by the single instruction **INC a**
- Rather than by a more obvious sequence that loads a into a register, adds one to the register and then stores the result back into a:

```
LD R0, a           // R0 = a  
ADD R0, R0, #1    // R0 = R0 + 1  
ST a, R0          // a = R0
```

# The Target Language

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator
- We shall use as a target language assembly code for a simple computer that is representative of many register machines

# A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations and conditional jumps
- The underlying computer is a byte-addressable machine with  $n$  general-purpose registers  $R_0, R_1, \dots, R_{n-1}$
- A full-fledged assembly language would have scores of instructions
- We use a very limited set of instructions and assume that all operands are integers
- Most instructions consists of an operator followed by a target followed by a list of source operands
- A label may precede an instruction

# A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Load Operations:** The instruction **LD dst, addr** loads the value in location addr into location dst
  - This instruction denotes the assignment **dst = addr**
  - The most common form of this instruction is **LD r, x** which loads the value in location x into register r
  - An instruction of the form **LD r1, r2** is a register-to-register copy in which the contents of register r2 are copied into register r1

# A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Store Operations:** The instruction **ST x, r** stores the value in register r into the location x
  - This instruction denotes the assignment **x = r**

# A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Computation Operations** of the form **OP dst, src1, src2** where OP is a operator like ADD or SUB and dst, src1, and src2 are locations, not necessarily distinct.
  - The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2 and place the result of this operation in location dst
  - For example, **SUB r1, r2, r3** computes **r1 = r2 - r3**
  - Any value formerly stored in r1 is lost but if r1 is r2 or r3, the old value is read first
  - Unary operators that take only one operand do not have a src2

# A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Unconditional jumps:** The instruction **BR L** causes control to branch to the machine instruction with label L (BR stands for branch)

# A Simple Target Machine Model(Cont...)

We assume the following kinds of instructions are available:

- **Conditional jumps** of the form **Bcond r, L** where r is a register, L is a label and cond stands for any of the common tests on values in the register r
  - For example, **BLTZ r, L** causes a jump to label L if the value in register r is less than zero and allows control to pass to the next machine instruction if not.

# A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a **variable name**  $x$  referring to the memory location that is reserved for  $x$  (that is l-value of  $x$ )
- A location can also be an indexed address of the form  $a(r)$  where  $a$  is a variable and  $r$  is a register
- The memory location denoted by  $a(r)$  is computed by taking the l-value of  $a$  and adding to it the value in register  $r$

# A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- For example, the instruction **LD R1, a(R2)** has the effect of setting **R1 = contents(a +contents(R2))** where **contents(x)** denotes the contents of the register or memory location represented by x
- This addressing mode is useful for accessing arrays, where a is the base address of the array (that is the address of the first element) and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

# A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- A memory location can be an **integer indexed by a register**
- For example, **LD R1, 100(R2)** has the effect of setting **R1 = contents(100+contents(R2))** that is of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2

# A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- We also allow two **indirect addressing modes**
- **\*r** means the memory location found in the location represented by the contents of register r and
- **\*100(r)** means the memory location found in the location obtained by adding 100 to the contents of r
- For example, **LD R1, \*100(R2)** has the effect of setting **R1 = contents(contents(100+contents(R2)))** that is of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2

# A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- Finally, we allow an **immediate constant addressing mode**
- The constant is prefixed by #
- The instruction **LD R1, #100** loads the integer 100 into register R1 and
- **ADD R1, R1, #100** adds the integer 100 into register R1
- Comments at the end of instructions are preceded by //

## Example 8.2

- The three-address statement  $x = y - z$  can be implemented by the machine instructions:

<b>LD R1, y</b>	// R1 = y
<b>LD R2, z</b>	// R2 = z
<b>SUB R1, R1, R2</b>	// R1 = R1 - R2
<b>ST x, R1</b>	// x = R1

- One of the goals of a good code-generation algorithm is to avoid using all four of these instructions whenever possible
- For example, y and/or z may have been computed in a register and if so we can avoid the LD step(s)
- Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed

## Example 8.2 (Cont...)

- Suppose **a** is an array whose elements are 8-byte values
- Also assume elements of **a** are indexed starting at 0
- We may execute the three-address instruction **b = a [i]** by the machine instructions:

```
LD R1, i           // R1 = i
MUL R1, R1, 8     // R1 = R1 * 8
LD R2, a(R1)       // R2 = contents (a + contents (R1))
ST b, R2          // b = R2
```

## Example 8.2(Cont...)

- Similarly, the assignment into the array a represented by three-address instruction **a[j] = c** is implemented by:

<b>LD R1, c</b>	// R1 = c
<b>LD R2, j</b>	// R2 = j
<b>MUL R2, R2, 8</b>	// R2 = R2 * 8
<b>ST a(R2), R1</b>	// contents (a + contents (R2)) = R1

## Example 8.2 (Cont...)

- To implement a simple pointer indirection, such as the three-address statement **x = \*p**, we can use machine instructions like:

**LD R1, p**

// R1 = p

**LD R2, 0(R1)**

// R2 = contents (0 + contents (R1))

**ST x, R2**

// x = R2

## Example 8.2 (Cont...)

- The assignment through a pointer `*p = y` is similarly implemented in machine code by:

<b>LD R1, p</b>	<code>// R1 = p</code>
<b>LD R2, y</b>	<code>// R2 = y</code>
<b>ST 0(R1), R2</b>	<code>// contents (0 + contents (R1)) = R2</code>

## Example 8.2(Cont...)

- A conditional jump three-address instruction like **if x < y goto L**
- The machine-code equivalent would be something like:

<b>LD R1, x</b>	// R1 = x
<b>LD R2, y</b>	// R2 = y
<b>SUB R1, R1, R2</b>	// R1 = R1 - R2
<b>BLTZ R1, M</b>	// if R1 < 0 jump to M

- Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L. As for any three-address instruction we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored.

# Program and Instruction Costs

- We often **associate a cost** with compiling and running a program
- Depending on what aspect of a program we are interested in optimizing, some common cost measures are the **length of compilation time and the size, running time and power consumption** of the target program
- Determining the actual cost of compiling and running a program is a complex problem
- Finding an optimal target program for a given source program is an undecidable problem in general and many of the sub problems involved are **NP-hard**.
- As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

## Program and Instruction Costs (Cont...)

- We shall **assume** each target-language instruction has an associated cost
- For simplicity, we take the **cost of an instruction** to be **one plus** the costs associated with the **addressing modes of the operands**
- This cost corresponds to the length in words of the instruction
- **Addressing modes involving registers have zero additional cost and**
- **Modes involving a memory location or constant in them have an additional cost of one because such operands have to be stored in the words following the instruction.**

# Program and Instruction Costs (Cont...)

Some examples:

- **LD R0, R1**
  - The instruction **LD R0, R1** copies the contents of register R1 into register R0
  - This instruction has a **cost of one** because no additional memory words are required
  
- **LD R0, M**
  - The instruction **LD R0, M** loads the contents of memory location M into register R0
  - The **cost is two** since the address of memory location M is in the word following the instruction

# Program and Instruction Costs (Cont...)

Some examples:

- **LD R1, \*100(R2)**
  - The instruction **LD R1, \*100(R2)** loads into register R1 the value given by **contents(contents(100+contents(R2)))**
  - The **cost is three** because the constant 100 is stored in the word following the instruction
- Cost of a target-language program on a given input is
  - the sum of costs of the individual instructions executed when the program is run on that input
  - Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs.

# Addresses in the Target Code

- Names in the **IR** can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation.
- Each executing program runs in its own logical address space that was partitioned into **four code and data areas**:
  - ✓ A statically determined area **Code** that holds the executable target code. The size of the target code can be determined at compile time.
  - ✓ A statically determined data area **Static** for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

## Addresses in the Target Code (Cont...)

- ✓ A dynamically managed area **Heap** for holding data objects that are allocated and freed during program execution. The size of the Heap cannot be determined at compile time.
- ✓ A dynamically managed area **Stack** for holding activation records as they are created and destroyed during procedure calls and returns. Like the Heap, the size of the Stack cannot be determined at compile time.

# Static Allocation

- To illustrate code generation for simplified procedure calls and returns we shall focus on the following three-address statements:
  - **call callee**
  - **return**
  - **halt**
  - **action**, which is a placeholder for other three-address statements

## Static Allocation (Cont...)

- The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table
- We shall first illustrate how to store the return address in an activation record on a procedure call and how to return control to it after the procedure call
- For convenience, we assume the first location in the activation holds the return address.

## Static Allocation (Cont...)

- Let us consider the code needed to implement the simplest case static allocation
- Here, a **call callee** statement in the intermediate code can be implemented by a sequence of two target-machine instructions:  
**ST callee.staticArea, #here + 20**  
**BR callee.codeArea**
- The **ST** instruction saves the return address at the beginning of the activation record for callee and
- The **BR** transfers control to the target code for the called procedure callee

# Static Allocation (Cont...)

**ST callee.staticArea, #here + 20**

**BR callee.codeArea**

- The attribute **callee.staticArea** is a constant that gives the address of the beginning of the activation record for callee and
- The attribute **callee.codeArea** is a constant referring to the address of the first instruction of the called procedure callee in the Code area of the run-time memory.
- The operand **#here + 20** in the ST instruction is the literal return address
- It is the address of the instruction following the BR instruction

# Static Allocation (Cont...)

**ST callee.staticArea, #here + 20**

**BR callee.codeArea**

- We assume that **#here** is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of **5 words or 20 bytes**
- The code for a procedure ends with a return to the calling procedure
- Except that the first procedure has no caller, so its final instruction is **HALT** which returns control to the operating system
- A **return callee** statement can be implemented by a simple jump instruction **BR \*callee.staticArea** which transfers control to the address saved at the beginning of the activation record for callee.

## Example 8.3

Suppose we have the following three-address code:

// code for c

action1

call p

action2

halt

// code for p

action3

return

## Example 8.3 (Cont...)

- We use the pseudo-instruction **ACTION** to represent the sequence of machine instructions to execute the statement action
- This represents three-address code that is not relevant for this discussion
- We arbitrarily start the **code** for **procedure c** at address **100** and for **procedure p** at address **200**
- We assume that each **ACTION** instruction takes **20 bytes**
- We further assume that the **activation records** for these procedures are **statically allocated** starting at locations **300** and **364** respectively

## Example 8.3 (Cont...)

// code for c	// code area for c:	
action1	100:	ACTION1 // code for activation
call p	120:	ST 364, #140 // save return address 140 in location 364
action2	132:	BR 200 // call p
halt	140:	ACTION2 // code for activation
// code for p	160:	HALT // return to operating system
action3		
return		

### Example 8.3(Cont...)

	<b>// code area for p:</b>	
<b>// code for c</b>	<b>200: ACTION3</b>	<b>// code for activation</b>
<b>action1</b>	<b>220: BR *364</b>	<b>// return to address saved in location 364</b>
<b>call p</b>		
<b>action2</b>		
<b>halt</b>	<b>// static area for c:</b>	<b>// 300-363 hold activation record for c</b>
<b>// code for p</b>	<b>300:</b>	<b>// return address</b>
<b>action3</b>	<b>304:</b>	<b>// local data for c</b>
<b>return</b>		
	<b>// static area for p:</b>	<b>// 364-451 hold activation record for p</b>
	<b>364: 140</b>	<b>// return address</b>
	<b>368:</b>	<b>// local data for p</b>



# Stack Allocation

- **Static** allocation can become **stack** allocation by using **relative addresses** for storage in activation records.
- In stack allocation the position of an activation record for a procedure is not known until run time.
- This **position** is usually stored in a **register (SP)**, so words in the activation record can be accessed as offsets from the value in this register.
- The **indexed address mode** of our target machine is convenient for this purpose.

# Stack Allocation (cont..)

- When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
- After control returns to the caller, we decrement SP, thereby deallocating the activation record of the called procedure.
- **Step-1:** First procedure initializes the stack by setting SP to the start of the stack area in memory:

```
LD SP, #stackStart    // initialize the stack
// code for the first procedure
HALT                  // terminate execution
```

# Stack Allocation (cont..)

- **Step-2:** A procedure call (**call callee**) sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD SP, SP, #caller.recordSize // increment stack pointer  
ST 0(SP), #here + 16        // save return address  
BR callee.codeArea           // jump to the callee
```

# Stack Allocation (cont..)

- **Step-3:** The return (**return**) sequence consists of two parts.
  - Part-1: The called procedure transfers control to the return address using

**BR \*0(SP)** // return to caller

- Part-2: The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

**SUB SP, SP, #caller.recordSize** // decrement stack pointer

## Stack Allocation (cont..) Example-8.4

- Consider the pseudo code of quicksort again.
- Suppose that the sizes of the activation records for procedures **m**, **p**, and **q** have been determined to be **msize**, **psize**, and **qsize**, respectively.
- The **first word** in each **activation record** will hold a **return address**.
- We arbitrarily assume that the code for these **procedures starts** at addresses 100, 200, and 300, respectively, and that the **stack** starts at address **600**.
- We assume that ACTION4 contains a conditional jump to the address 456 of the return sequence from **q**; otherwise, the recursive procedure **q** is condemned to call itself forever.
- Let **msize**, **psize**, and **qsize** be 20, 40, and 60, respectively

## Stack Allocation (cont..) Example-8.4

```
// code for m  
action1  
call q  
action2  
halt  
// code for p  
action3  
return  
// code for q  
action4  
call p  
action5  
call q  
action6  
call q  
return
```

# Stack Allocation (cont.)

## Example-8.4

```
// code for m  
action1  
call q  
action2  
halt  
// code for p  
action3  
return
```

```
// code for q  
action4  
call p  
action5  
call q  
action6  
call q  
return
```

```
100: LD SP, #600          // code for m  
108: ACTION1           // initialize the stack  
128: ADD SP, SP, #msize  // code for action1  
136: ST 0(SP), #152     // call sequence begins  
144: BR 300              // push return address  
152: SUB SP, SP, #msize  // call q  
160: ACTION2           // restore SP  
180: HALT                ...  
200: ACTION3           // code for p  
220: BR *0(SP)          // return  
300: ACTION4           // code for q  
320: ADD SP, SP, #qsize  // contains a conditional jump to 456  
328: ST 0(SP), #344     // push return address  
336: BR 200              // call p  
344: SUB SP, SP, #qsize  // call q  
352: ACTION5           ...  
372: ADD SP, SP, #qsize  // push return address  
380: ST 0(SP), #396     // call q  
388: BR 300              // call q  
396: SUB SP, SP, #qsize  // push return address  
404: ACTION6           // call q  
424: ADD SP, SP, #qsize  // push return address  
432: ST 0(SP), #440     // call q  
440: BR 300              // call q  
448: SUB SP, SP, #qsize  // return  
456: BR *0(SP)          // stack starts here
```

# **Basic Blocks and Flow Graphs**

# Basic Blocks and Flow Graphs

The representation is constructed as follows:

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the **properties** that
  - a) The **flow of control** can only **enter** the basic block through the **first instruction** in the block. That is, there are no jumps into the middle of the block.
  - b) **Control** will **leave** the block without **halting** or **branching**, except possibly at the last instruction in the block.
2. The **basic blocks** become the **nodes** of a **flow graph**, whose edges indicate which blocks can follow which other blocks.

# Basic Blocks

- Our first job is to partition a sequence of three-address instructions into basic blocks.

**Algorithm 8.5:** Partitioning three-address instructions into basic blocks.

**INPUT:** *A sequence of three-address instructions.*

**OUTPUT:** *A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.*

**METHOD:**

**First**, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader.

**Then**, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

# Basic Blocks

- The rules for finding leaders are:
  1. The **first three-address instruction** in the intermediate code is a **leader**.
  2. Any **instruction** that is the **target** of a conditional or unconditional jump is a **leader**.
  3. Any instruction that **immediately follows** a conditional or unconditional jump is a leader.

## Basic Blocks (Figure. 8.7)

- The **leaders** are instructions 1, 2, 3, 10, 12, and 13.
- The basic block of each leader contains all the instructions from itself until just before the next leader.
- The **basic block** of 1 is just 1, for leader 2 the block is just 2.
- Leader 3 has a basic block consisting of instructions 3 through 9, inclusive.
- Instruction 10's block is 10 and 11
- Instruction 12's block is just 12, and
- Instruction 13's block is 13 through 17.

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

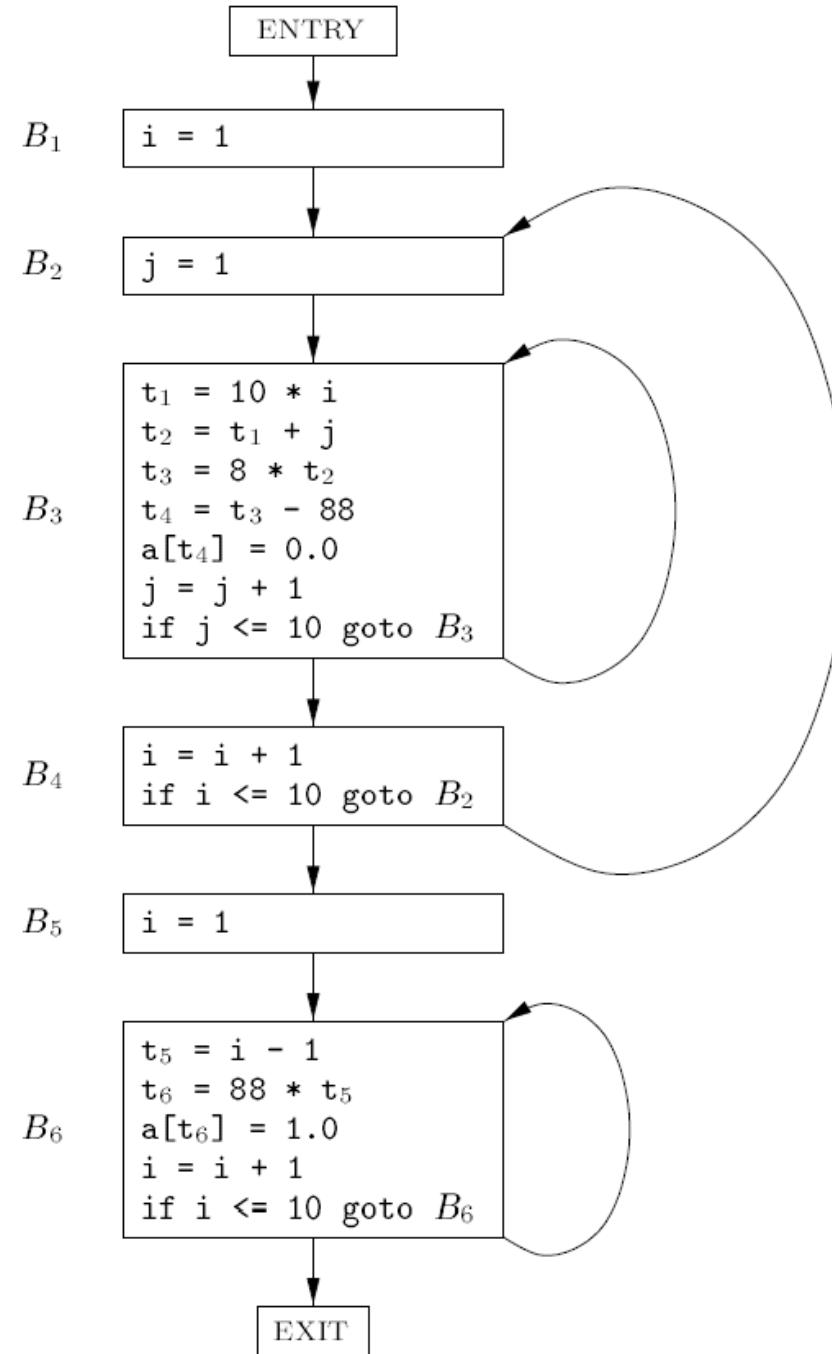
Figure. 8.7

# Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The **nodes** of the flow graph are the **basic blocks**.
- There is an **edge** from block **B** to block **C** if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.
- There are **two ways** that such an **edge** could be justified:
  - There is a conditional or unconditional jump from the end of B to the beginning of C.
  - C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

# Flow Graphs

Figure 8.9: Flow graph from Fig. 8.7



# Summary

- Code generation is the final phase of a compiler.
- Instruction selection
- Register allocation is the process of deciding which IR values to keep in registers. Register assignment is the process of deciding which register should hold a given IR value.
- CISC and RISC machine intro
- Target machine model with instruction and program cost
- Address in the target code using static and stack allocation
- A basic block is a maximal sequence of consecutive three-address statements in which flow of control can only enter at the first statement of the block and leave at the last statement without halting or branching except possibly at the last statement in the basic block.
- Flow graph is a graphical representation of a program in which the nodes of the graph are basic blocks and the edges of the graph show how control can flow among the blocks.