5.1 Syntax-Directed Translation (SDT)

- Syntax- directed translation is a grammar-oriented compiling technique that translates programming language constructs that are guided by CFG into an intermediate representation.

- Almost all compilers are syntax directed. i.e the compilation process is driven by the syntactic structure of a source program.

- This phase of the compiler is considered to be semantic analysis that computes the additional information needed for compilation once the syntactic structure of a program is known

- We associate information with a programming language construct by attaching attributes to the grammar symbols of the construct.

- We can augment grammar with information to control semantic analysis and translation

- The most general approach to syntax-directed translation is to construct a parse tree or a syntax tee and then to compute values at the nodes of the tree by visiting the nodes of the tree.

There are two translation techniques :

1) Syntax-Directed Definition(SDD)
2) Syntax-Directed Translation (SDT) scheme

- Between two notations, SDD is more readable and hence more useful.

Syntax-Directed Definition (SDD)

- A syntax-directed definition (SDD) is a context free grammar together with attributes and rules.

- Attributes are associated with grammar symbols and semantic rules are associated with productions.

- A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar 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:** An attribute is any quantity associated with a programming construct.

It may be of any kind: numbers, types, value of an expression, number of instructions, strings, memory location, table references.

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

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

Inherited and Synthesized Attributes

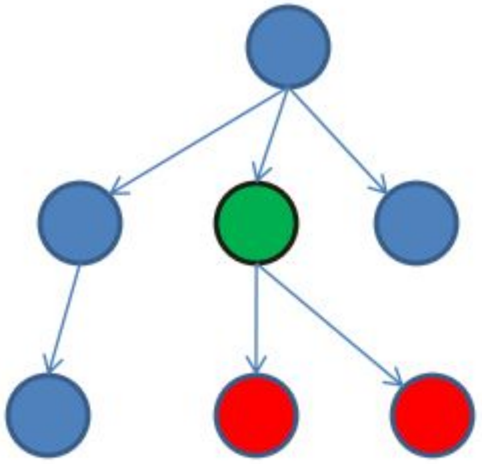
There are two types of attributes for nonterminals:

1) Synthesized attribute:

- A ***synthesized attribute*** for a non terminal 'A' at a parse-tree node N is defined by a semantic rule associated with the production at N. The production must have 'A' at 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.

2) 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.

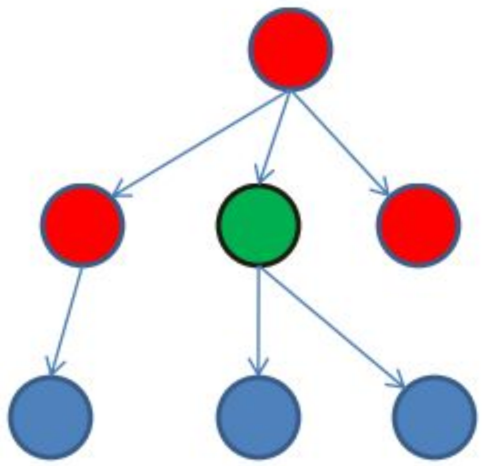


SDD involving only synthesized attributes is called ***S-attributed***

Synthesized Attributes

Attribute of the node is defined in terms of:

- Attribute values at children of the node
- Attribute value at node itself



Inherited Attributes

Attribute of the node is defined in terms of:
- Attribute values at parent of the node
- Attribute values at siblings
- Attribute value at node itself

- Nonterminals can have both synthesized and inherited attributes.
- Terminals can have only 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.

Syntax-Directed Definition –of a simple desk calculator

Production	**Semantic Rules**
$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

- It evaluates expression terminated by an endmarker **n**.

- Each of the nonterminals E, T, and F are associated with a single synthesized attribute *val*. Rule for production 1, L → E **n** sets L.val to E.val , which we shall see is the numerical value of the entire expression.

- The token(terminal) **digit** has a synthesized attribute *lexval* (it is an integer valued returned by the lexical analyzer).

- An SDD that involves only synthesized attributes is called S-attributed.
- In an S-attributed SDD each rule computes an attribute for the nonterminal at the head of a production from attributes from the body of the production.
- For simplicity, the examples have semantic rules without side effects.
- (side effects- such as printing the result computed by the desk calculator or interacting with the symbol table)
- **An SDD without side effects is sometimes called attribute grammar.**

Evaluating an SDD at the Nodes of a Parse tree

- The rules of the SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.

- A parse tree, showing the values of its attributes is called an **Annotated parse tree.**

How do we construct an annotated parse tree? 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.
  - Ex: If all the attributes are **synthesized**, then we must evaluate the '*Val'* attributes at all the children of a node before we can evaluate the 'V*al'* attributes at the node itself.

Annotated Parse Tree

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

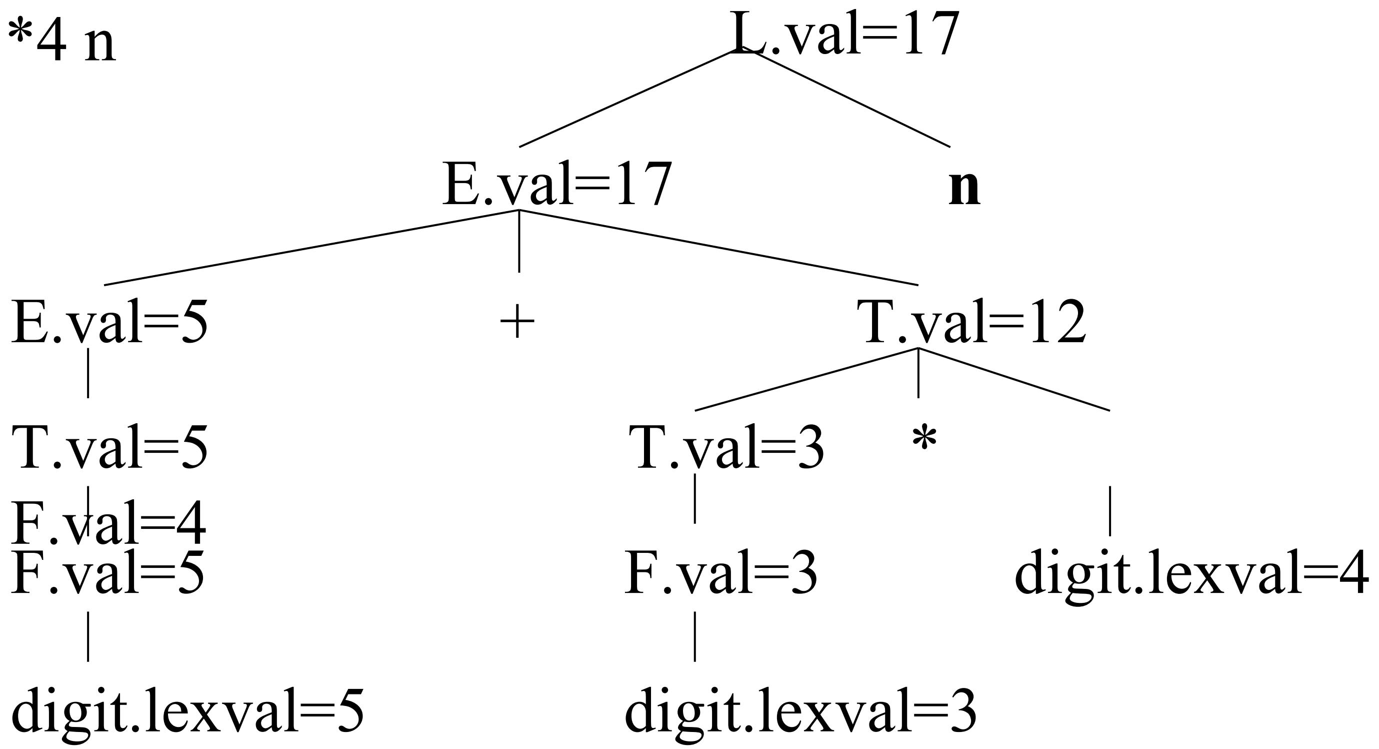
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

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

- SDD With synthesized attributes, we can evaluate the attributes in any bottom up order, such as postorder traversal
- For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate atrributes at nodes.

Annotated Parse Tree -- Example

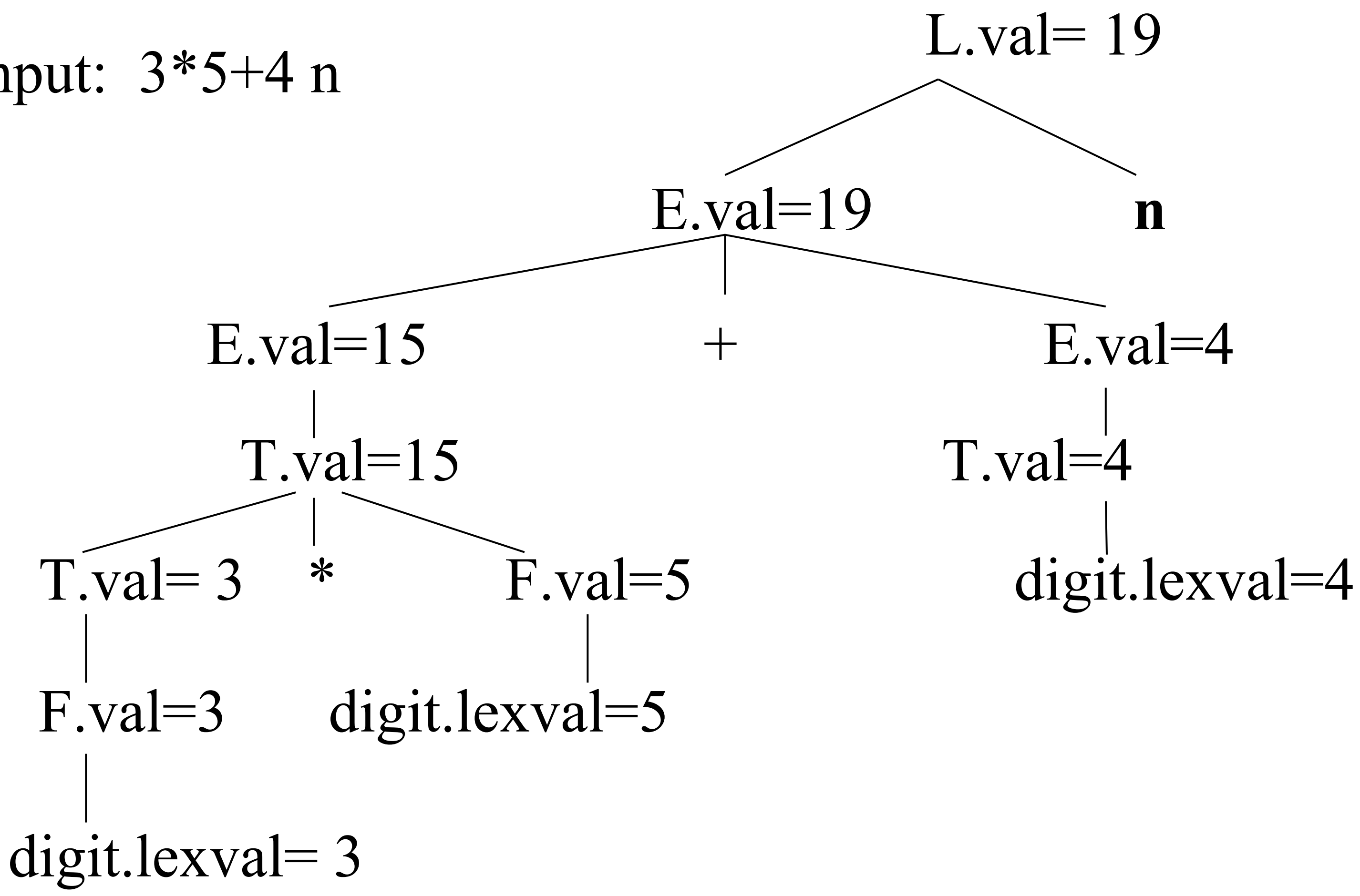
Input: 5+3*4 n



Each node for nonterminal has attribute *val* computed in a bottom – up order

Annotated Parse Tree -- Example

Input: 3*5+4 n

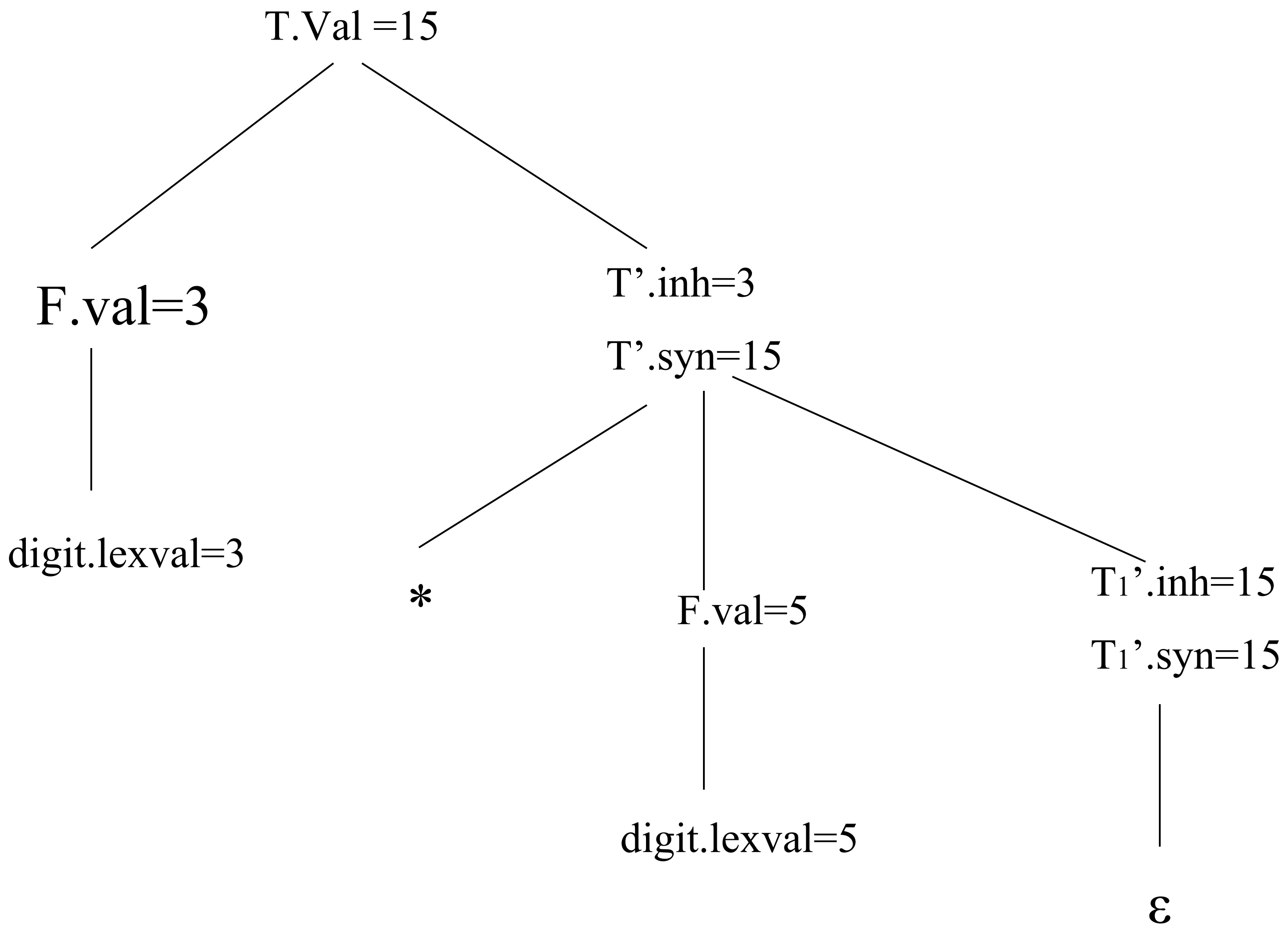


SDD based on a grammar suitable for top-down parsing

Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn=T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

The semantic rules are based on the idea that the left operand of the operator * is inherited. More precisely, the head T' of the production $T' \rightarrow *FT_1'$ inherits the left operand of * in the production body.

Annotated Parse Tree for 3 * 5



Exercise Questions

1. Using SDD for simple desk calculator, write annotated parse tree for (3+4)*(5+6)

2. Write annotated parse tree for expression
   a) 1*2*3*(4+5)n
   b) (9+8*(7+6)+5)*4n

5.2 Evaluation Orders for SDD's

- **Dependency graphs** – are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.

- While an annotated parse tree shows the value of attributes, a dependency graph helps us to determine how these values can be computed.

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

Dependency graphs (Contd…)

1. 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.

2. If a semantic rule associated with a production p defines the value of **synthesized attribute** A.b in terms of the value of X.c. Then the dependency graph has an edge from X.c to A.b .

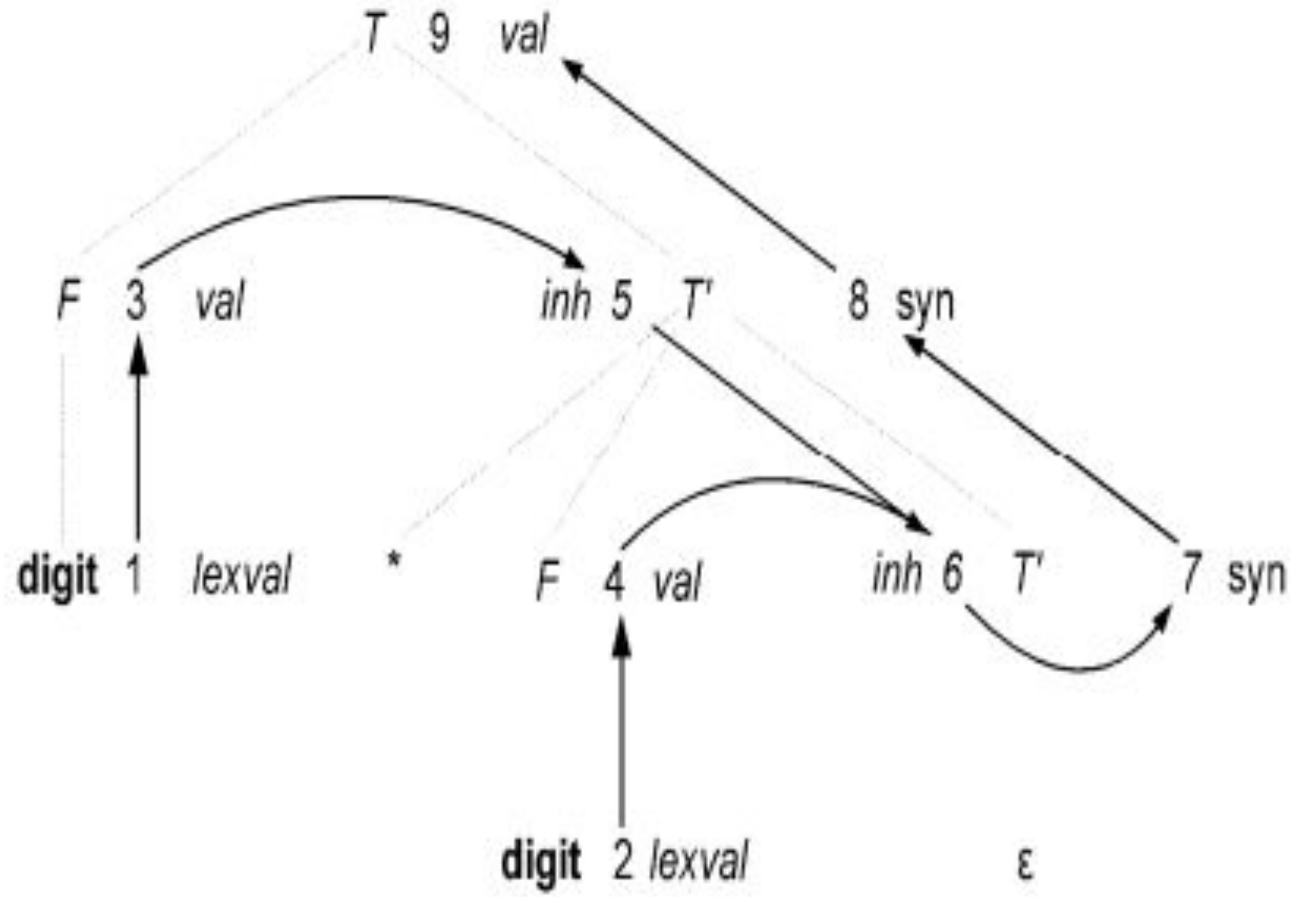
3. If a semantic rule associated with a production p defines the value of **inherited attribute** B.c in terms of the value X.a. Then , the dependency graph has an edge from X.a to B.c.

- Consider the following production and rule:

Production	**Semantic Rule**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

At every node N labeled E, with the children corresponding to the body of this production, the synthesized attribute *val* at N is computed using the values of *val* at the two children labeled E and T.

Dependency graph for annotated parse tree for 3*5



Ordering the evaluation of attributes

- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of the parse tree.

- If dependency graph has an edge from node M to node N then attribute corresponding to M must be evaluated before the attribute of N

- Thus the only allowable orders of evaluation are those sequence of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge 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 a graph

- 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 then there is always at least one topological sort.
- One topological sort for the previous graph is 1,2,3,4,5,6,7,8,9
- Other is 1,3,5,2,4,6,7,8,9

Classes of SDD

- Syntax-directed definitions are used to specify syntax-directed translations.
- These translations can be implemented using two classes of SDD’s, that guarantee an evaluation order.
- These two classes can be implemented efficiently with top-down or bottom-up parsing.
- We can evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).

1) S-Attributed Definitions:

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

- Eg: any bottom – up approach based SDD

i.e SDD based on grammar for arithmetic expression using bottom up approach

- When an SDD is S-attributed, we can evaluate its attributes in bottom – up order of the nodes of the parse tree.

- It is simple to evaluate the attributes in an S-attributed SDD by performing a postorder traversal of the parse tree.

- We apply function postorder to the root of the parse tree. i.e

```
postorder(N)
{
     for(each child C of N, from the left)
          postorder(C);
     evaluate the attributes associated with node N;
}
```

- Postorder traversal corresponds exactly to the order in which an LR parser reduces a production body to its head.

2) L- Attributed Definitions:

The idea behind this class is that, between the attributes associated with the production body , dependency-graph edges can go from left to right but not from right to left.

A SDD is L-attributed if each attribute is:

1)Synthesized or

2)Inherited, but with the rules as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$ and there is a inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

a) Inherited attributes associated with the head A

b) Either inherited or synthesized attributes associated with the occurrence of symbols $X_1, X_2 \ldots X_{i-1}$ located to the left of X_i.

c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but in such a way that there are no cycles in dependency graph formed by attributes of X_i

- Eg for L atttributed SDD

SDD of arithmetic expression grammar using top-down approach . In each of the 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.

- Note: Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

- Consider the following SDD. Is it S-attributed or L-attributed?

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

- The first rule is either S attributed or L attributed SDD.
- It defines synthesized attribute A.s in terms of an attribute at a child.
- The second rule defines an inherited attribute B.i, The SDD cannot be L attributed because the attribute C.c is used to define B.i and C is to the right of B in the production body.

Semantic rules with controlled side effects

- Translations involve side effects: for eg- a desk calculator might print the result, a type declaration might enter the type of an identifier into the symbol table.

- Eg of side effect to the simple desk calculator. Let us modify the semantic rules as:

Production	**Semantic Rules**
1) L → E **n**	print(E.val)

- Semantic rules that are executed for their side effects, such as print(E.val) will be treated as the definition of dummy synthesized attributes associated with the head of the production. This modified SDD provides same translation as that of SDD without side effect, since the print statement is executed at the end.

Syntax-Directed Definition for simple type declaration

Production	**Semantic Rules**
1) $D \rightarrow T\ L$	L.inh = T.type
2) $T \rightarrow$ **int**	T.type = integer
3) $T \rightarrow$ **float**	T.type = **float**
4) $L \rightarrow L_1$, **id**	L_1.inh = L.inh addtype(**id**.entry,L.inh)
5) $L \rightarrow$ **id**	addtype(**id**.entry,L.inh)

- Nonterminal D represents a declaration in production 1 which consists of a basic type T followed by a list L of identifiers.
- For each identifier on the list, the type is entered into the symbol table entry for the identifier.
- T has one attribute, T.type which is a synthesized attribute.
- L has one attribute called inh to emphasize that it is an inherited attribute.

- Purpose of L.inh is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol table entries.

- This SDD will not check whether an identifier is declared more than once.

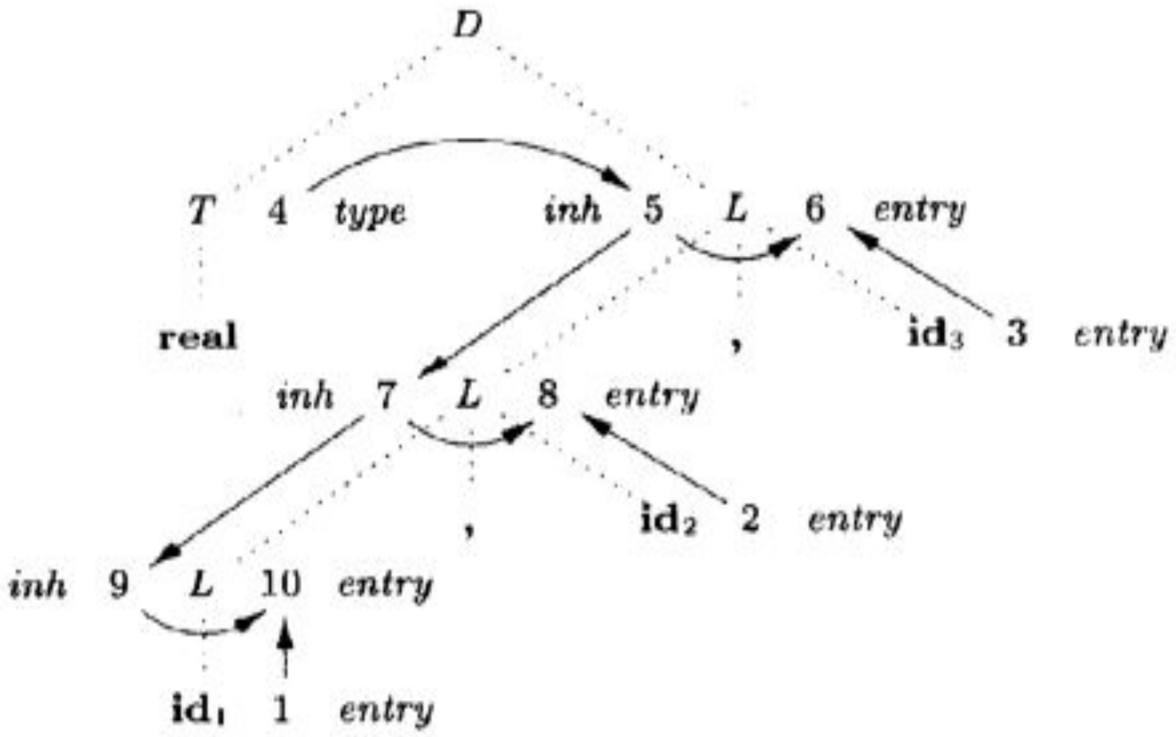
- Production 2 and 3 evaluate the synthesized attribute T.type

- This type is passed to L.inh, to pass it down to the list using production 4.

- The value L_1.inh is computed by copying value of L.inh from the parent of that node. i.e the head of that production.

- Production 4 and 5 have a rule in which function addType is called with two arguments:
  1)id.entry, a lexical value that points to a symbol-table object
  2)L.inh, the type being assigned to every identifier on the list.

- addType function installs the type L.inh as the type of the represented identifier.



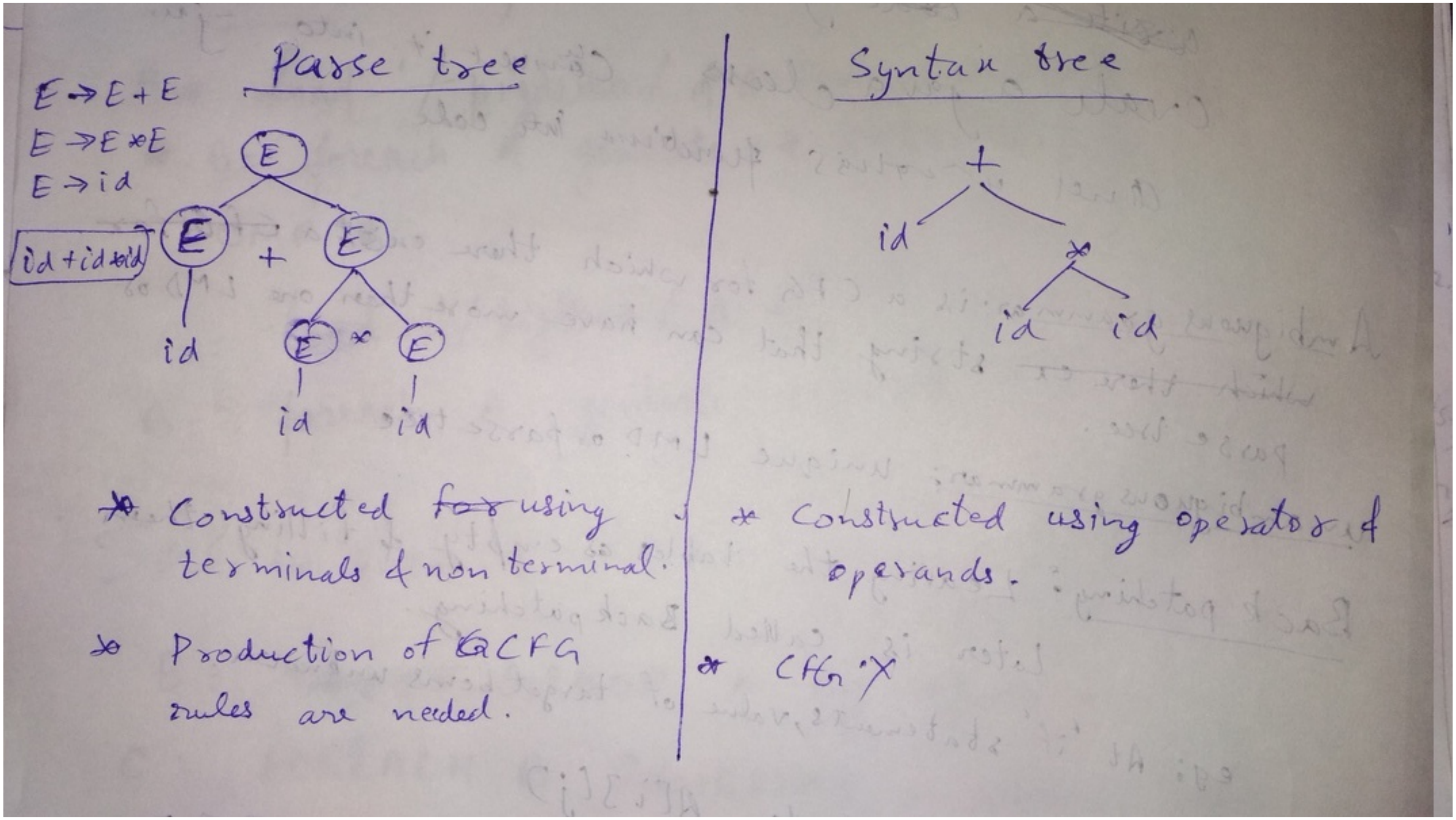
Dependency graph for the input string **float id$_1$, id$_2$, id$_3$**

Applications of Syntax-Directed Translation

The main application is **construction of syntax Trees**

- Some compilers use syntax trees as an intermediate representation instead of parse tree.
- Syntax tree is condensed form of parse tree.
- To complete the translation to intermediate code, the compiler will 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.
- We consider two SDD's for constructing syntax trees
  - First , an S-attributed definition, which is suitable for bottom-up parsing
  - The second, L-attributed definition which is suitable for top-down parsing

Difference between syntax tree and parse tree



Parse tree

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

$\boxed{id+id*id}$

$$\begin{tikzcd}
 & & E \arrow[dll] \arrow[d] \arrow[drr] & & \\
E \arrow[d] & & + & & E \arrow[dl] \arrow[d] \arrow[dr] \\
id & & & E \arrow[d] & * & E \arrow[d] \\
 & & & id & & id
\end{tikzcd}$$

* Constructed ~~for~~ using terminals & non terminal.

* Production of ~~a~~ CFG rules are needed.

Syntax tree

$$\begin{tikzcd}
 & + \arrow[dl] \arrow[dr] & \\
id & & * \arrow[dl] \arrow[dr] \\
 & id & & id
\end{tikzcd}$$

* Constructed using operator & operands.

* CFG ✗

Construction of Syntax trees

- Each node in the syntax tree represents a construct, and the children of the node represents the meaningful components of the construct.

- We shall implement the nodes of a syntax tree represented by objects with a suitable number of fields.

- Each object will have an ***op*** field that is the label of the node.

- The objects will have additional fields as follows:

  - If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function ***Leaf(op,val)*** creates a leaf object. If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.

  - 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 , c1,c2,…..ck) creates an object with first field ***op*** and ***k*** additional fields for the ***k*** children ***c1,c2,…..ck***

SDD for constructing syntax tree for simple expressions (S-attributed definition) during bottom-up parsing

Production	**Semantic Rules**
1) $E \rightarrow E_1 + T$	E.node = **new** Node ('+', E_1 .node, T.node)
2) $E \rightarrow E_1 - T$	E.node = **new** Node ('-', E_1 .node, T.node)
3) $E \rightarrow T$	E. node= T.node
4) $T \rightarrow (E)$	T.node = E.node
5) $T \rightarrow$ **id**	T.node= **new** Leaf (**id**, **id**.entry)
6) $T \rightarrow$ **num**	T.node= **new** Leaf (**num**, **num**.val)

- This is an example for S-attributed definition
- It involves only the binary operators + and – with same precedence and left associativity.
- All nonterminals have one synthesized attribute ***node***, which represents a node of the syntax tree.

- Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with'+' for op and two children, E_1 .node and T.node.

- Last two productions with leaf node, we construct *Leaf* to create a suitable node which becomes the value of T.node.

Construct a syntax tree for the input **a-4+c**

- First construct parse tree using dotted lines.

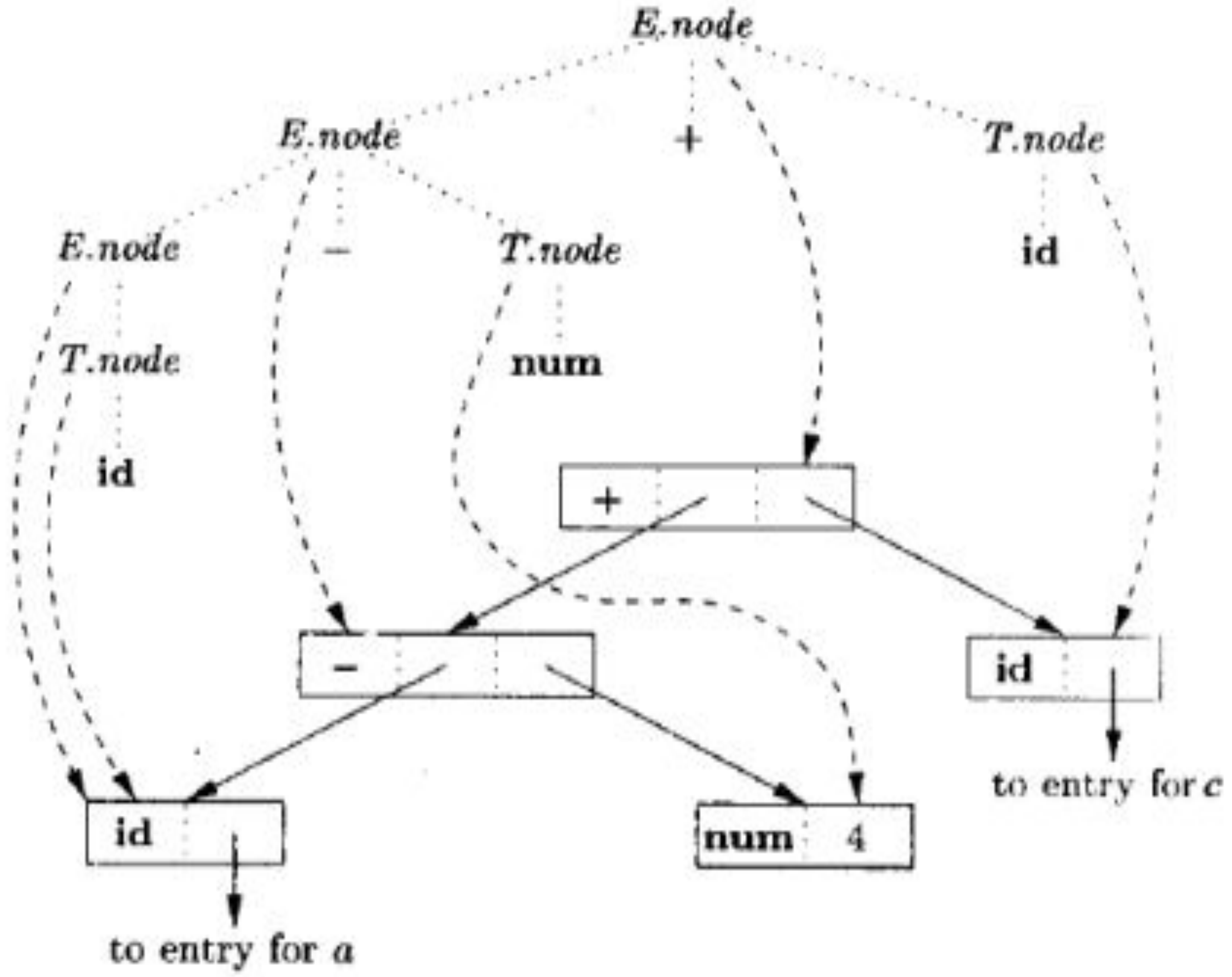
- According to the semantic rules start evaluation using bottom up approach from left to right.

- Nodes of the syntax tree are shown as records with op field first.

- Syntax-tree edges are drawn using solid line arrows.

- Dashed line arrows are used to represent the values of E.node and T.node, each line points to the appropriate syntax-tree node
- Note: dotted lines are edges of parse tree that is not actually to be constructed.

Syntax tree for a-4+c

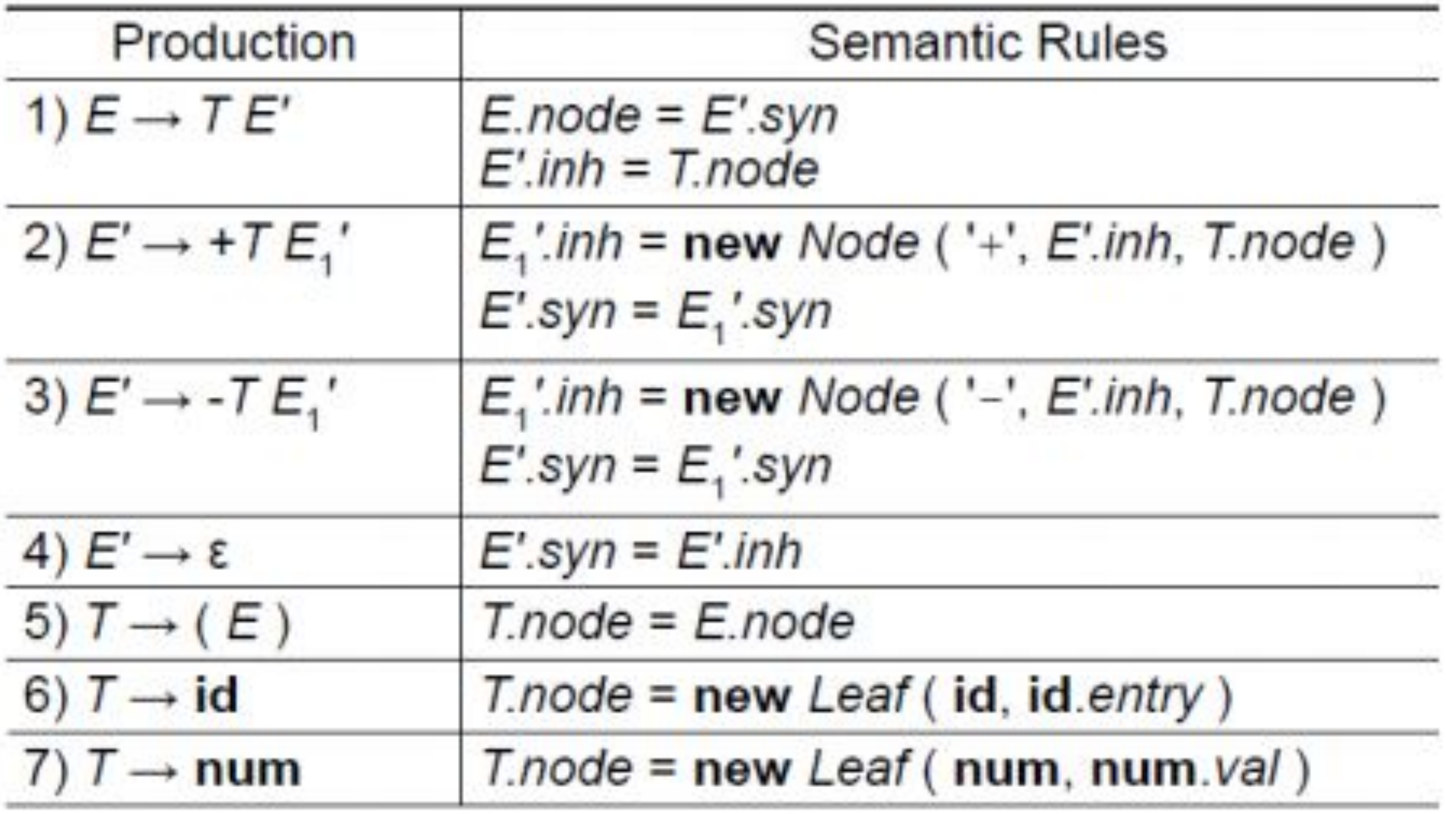


- Leaves a, 4,c are constructed by Leaf.
- Lexical value **id.entry** points into the symbol table and lexical value **num.val** is the numerical value of the constant.
- These leaves become the value of **T.node** at the parse tree nodes labeled T
- Rule 2 causes us to create a node with op equals the minus sign and pointers to the first two leaves.
- Production 1 produces the root node of the syntax tree by combining the node for – with the leaf

Steps in construction of syntax tree for a-4+c

1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('−', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

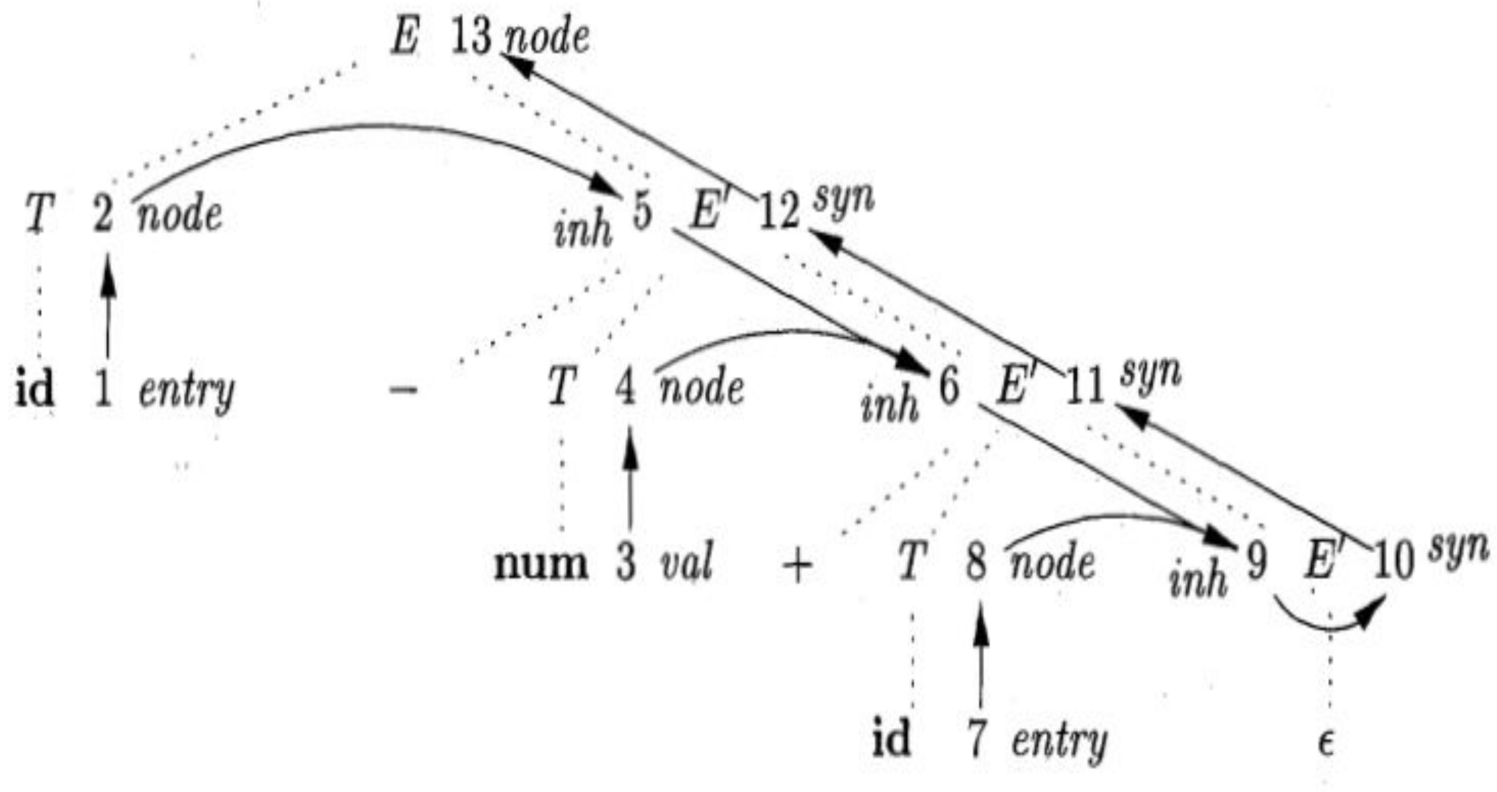
SDD for constructing syntax tree for simple expressions (L-attributed definition) during top – down parsing



Production	Semantic Rules
1) $E \rightarrow T E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \rightarrow +T E_1'$	$E_1'.inh =$ **new** *Node* ('+', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
3) $E' \rightarrow -T E_1'$	$E_1'.inh =$ **new** *Node* ('−', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
4) $E' \rightarrow \varepsilon$	$E'.syn = E'.inh$
5) $T \rightarrow (E)$	$T.node = E.node$
6) $T \rightarrow$ **id**	$T.node =$ **new** *Leaf* (**id**, **id**.*entry*)
7) $T \rightarrow$ **num**	$T.node =$ **new** *Leaf* (**num**, **num**.*val*)

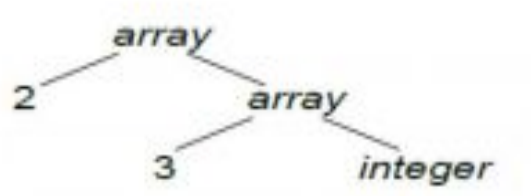
Construct syntax tree for a-4+c

Dependency graph for a-4+c with SDD of L-attributed

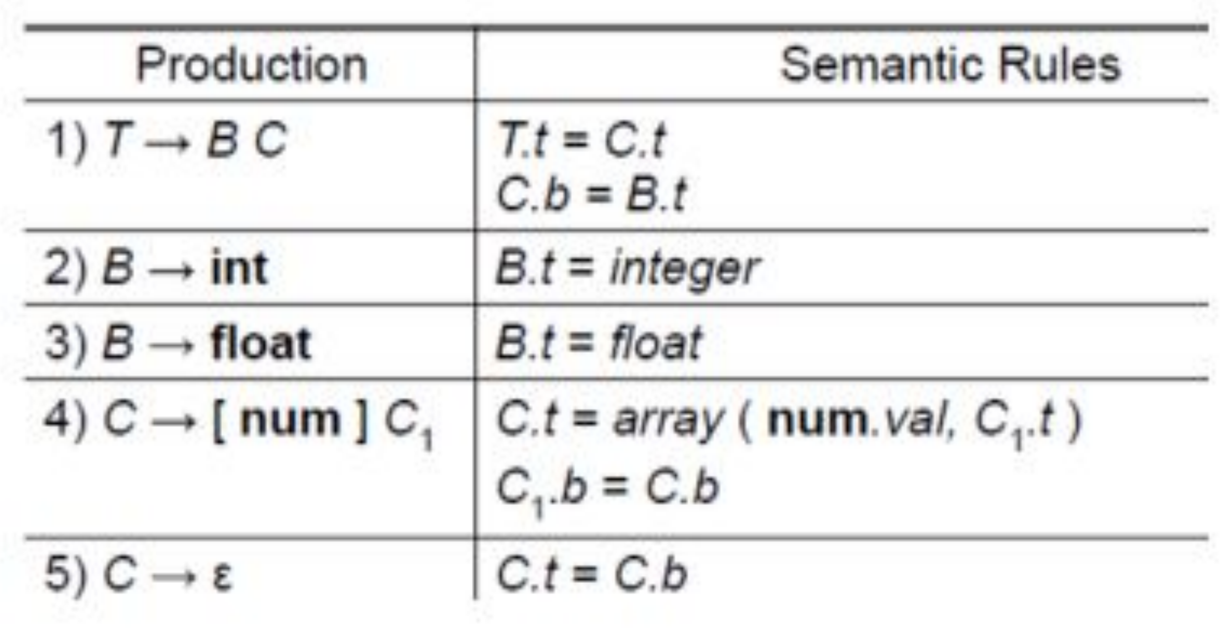


The Structure of a type

- Inherited attributes are useful when structure of the parse tree differs from the syntax of the input.
- Following example shows how the mismatch in the structure can be due to the design of the language.
- Consider the type in C, **int [2][3] which can be read as "array of 2 arrays of 3 integers"**
- This type expression **array(2, array(3, integer))** is represented by tree as follows:
- Operator array takes two parameters: a number and a type



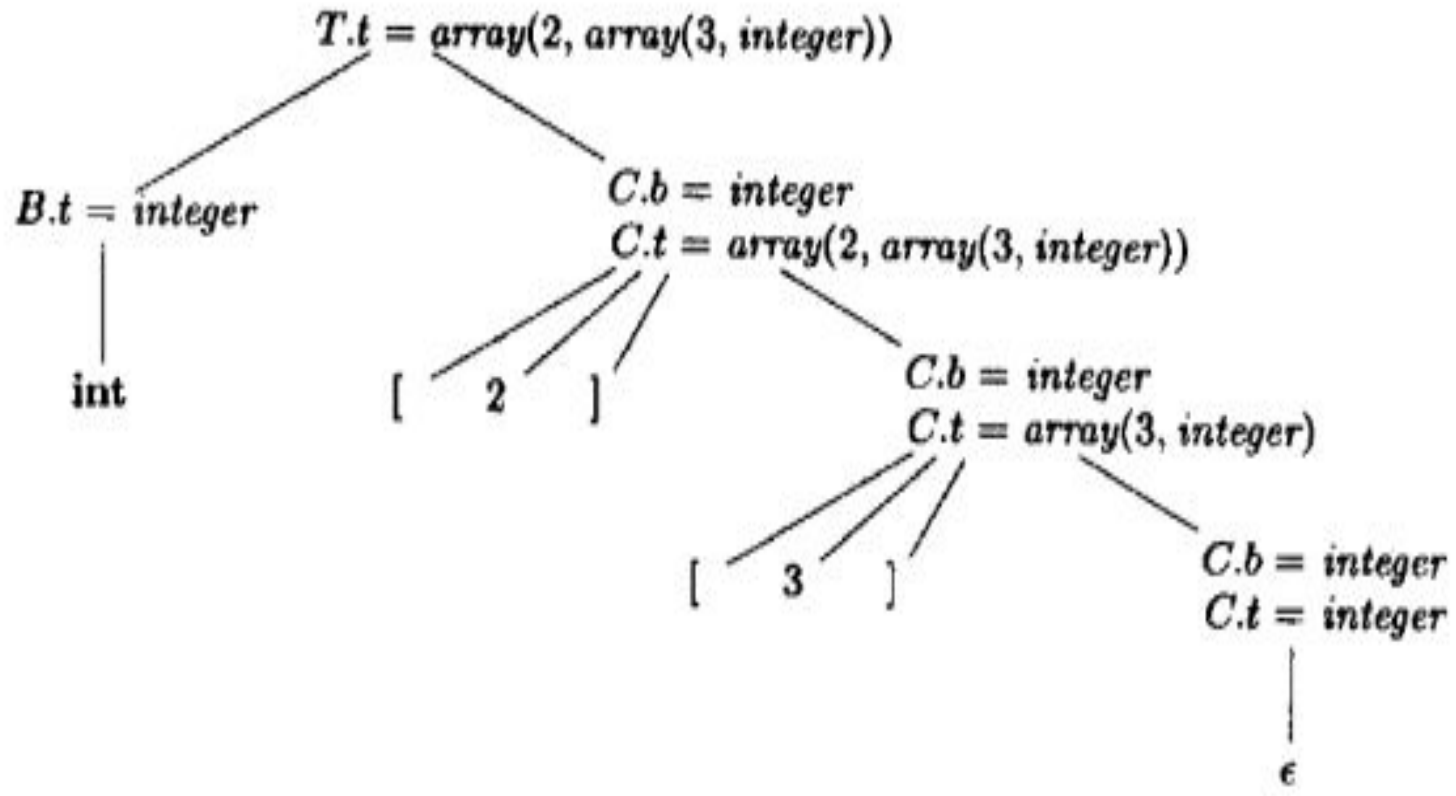
Type expression for **int[2][3]**



Production	Semantic Rules
1) $T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
2) $B \rightarrow$ **int**	$B.t = integer$
3) $B \rightarrow$ **float**	$B.t = float$
4) $C \rightarrow$ [**num**] C_1	$C.t = array($ **num**$.val, C_1.t)$ $C_1.b = C.b$
5) $C \rightarrow \varepsilon$	$C.t = C.b$

SDD that generates either basic type or array type

Annotated parse tree for the input string int[2][3]



- Non terminal B and T have a synthesized attribute ***t*** representing a type.
- Non terminal C has two attributes: an inherited attribute b and a synthesized attribute t.
- The inherited atrribute b pass a basic type down the tree and the synthesized t attributes accumulate the results