Syntax Directed Translation
Module 4

Prepared by

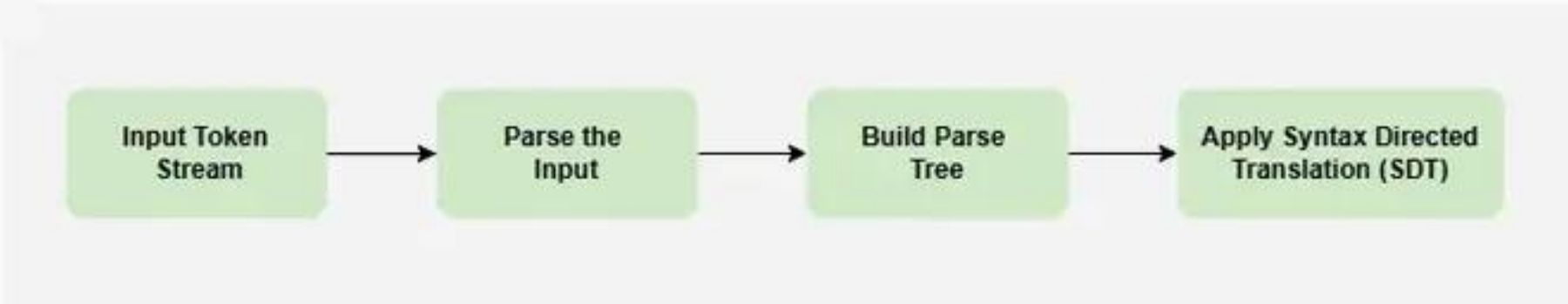
Jesna J S

Assistant Professor, CSE TKMCE

Syntax Directed Translation

- Syntax-Directed Translation (SDT) is a method used in compiler design to convert source code into another form while analyzing its structure.
- It integrates syntax analysis (parsing) with semantic rules to produce intermediate code, machine code, or optimized instructions.
- In SDT, each grammar rule is linked with semantic actions that **define how translation should occur**.
- These actions help in tasks like evaluating expressions, checking types, generating code, and handling errors.

- SDT ensures a systematic and structured way of translating programs, allowing information to be processed bottom-up or top-down through the parse tree.
- This makes translation efficient and accurate, ensuring that every part of the input program is correctly transformed into its executable form.



- SDT relies on three key elements:
  1. Lexical values of nodes (such as variable names or numbers).
  2. Constants used in computations.
  3. Attributes associated with non-terminals that store intermediate results.
- The general process of SDT involves constructing a parse tree or syntax tree, then computing the values of attributes by visiting its nodes in a specific order.
- *However, in many cases, translation can be performed directly during parsing, without explicitly building the tree.*

- There are **two notations for associating semantic rules** with productions
  1. Syntax-directed definitions
  2. Translation schemes.
- Syntax-directed definitions:
  - ✓High –level specifications for translations.
  - ✓Hide many implementation details and free from having to specify explicitly the order in which translation takes place.
- Translation schemes
  - ✓Indicate the order in which semantic rules are to be evaluated.
  - ✓So they allow some implementation details to be shown.

Conceptual view of SDT

- With both SDD and translation schemes do the following:
  1. **We parse the input token stream**
  2. **Build the parse tree**
  3. **Traverse the tree** as needed to evaluate the semantic rules at the parse-tree nodes.
  4. **Translation of the token stream is the result** obtained by evaluating the semantic rules.
- Evaluation of the semantic rules may
  - ❖Generate code
  - ❖Save information in a symbol table
  - ❖Issue error messages
  - ❖Perform any other activities

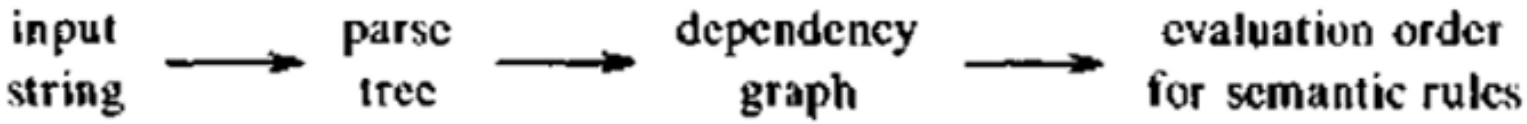


Fig. 5.1. Conceptual view of syntax-directed translation.

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attributes are attached to grammar symbols. **If A is a grammar symbol and a is its attribute then it is represented as A.a**
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.
- **To compute the value of attribute, we will assign semantic rules to a production.**

- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values, they can be broadly divided into two categories :
    1. synthesized attributes
    2. inherited attributes

- They are two types:

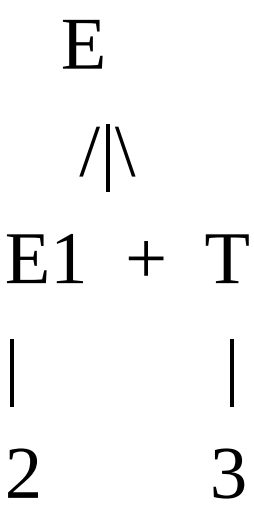
1. Synthesized attributes:
   - Defined by a semantic rule associated with the production at node N in the parse tree.
   - **Computed only using the attribute values of the children and the node itself.**
   - Mostly used in bottom-up evaluation

   Eg:

Grammar	*Semantic rule*
E → E1 + T	E.val = E1.val + T.val

Here val is a synthesized attribute because E.val is derived from its children E1.val and T.val.

- Parse tree for the string 2+3



- **Attribute Evaluation:**

  E1.val = 2

  T.val = 3

  E.val = E1.val + T.val = 2 + 3 = 5

- Synthesized attributes are evaluated in a **bottom-up manner**.

2. Inherited Attributes

✓Defined by a semantic rule associated with the parent production of node N.

✓**Computed using the attribute values of the parent, siblings, and the node itself.**

✓There are passed **downwards** from parent nodes to child nodes.

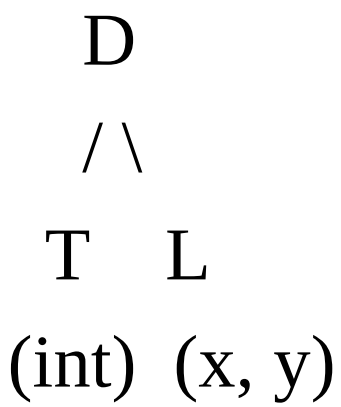
✓Used in top-down evaluation.

Example: ***Grammar Rule: Variable Declaration*** ***SDD Rule***

D→T L L.type = T.type

✓L.type is inherited because it is derived from its parent T.type.

- Syntax Tree for Input: **int x, y;**



- Attribute Evaluation:
  T.type = int
  L.type = T.type = int
- Inherited attributes are evaluated in a top-down manner.

Annotated Parse tree

- The **parse tree showing the values of attributes at each node** is called annotated parse tree.
- The process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

SYNTAX-DIRECTED DEFINITIONS

- SDD is a **generalization of a CFG** in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.
- For example, in an SDD, a production rule like: E → E1 + E2
- Might have a semantic rule that looks like : E.val = E1.val + E2.val
- SDDs can be classified into two main types based on when the semantic rules are applied:

1. S-attributed SDD:
   - The **semantic rules are applied at synthesized attributes** (which only depend on the children of a node in the syntax tree).
   - This means that the **value of a non-terminal can be computed based on the values of its children**.

2. L-attributed SDD:
   - The semantic rules are applied at inherited attributes (which can depend on both the children and the parent of a node in the syntax tree).
   - This is more general and allows for more complex semantic definitions.

Form of a Syntax-Directed Definition

In a syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$ where f is a function, and either

1. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production, or

2. b is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production.

S-attributed definition

- A SDD that uses synthesized attributes exclusively is said to be an S-attributed definition.
- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node, bottom up from the leaves to the root.
- **Computation Steps**
  1. *Write the SDD using appropriate semantic rules for each production in given grammar.*
  2. *The annotated parse tree is generated and attribute values are computed in bottom-up manner.*
  3. *The value obtained at root node is the final output.*

Example: A desk calculator that reads an input line containing an arithmetic expression involving digits, parentheses, the operators + and *, followed by a newline character n, and prints the value of the expression

Production	Semantic Rules
$L \rightarrow En$	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 * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

Production	Semantic Rules
$L \rightarrow En$	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 * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval

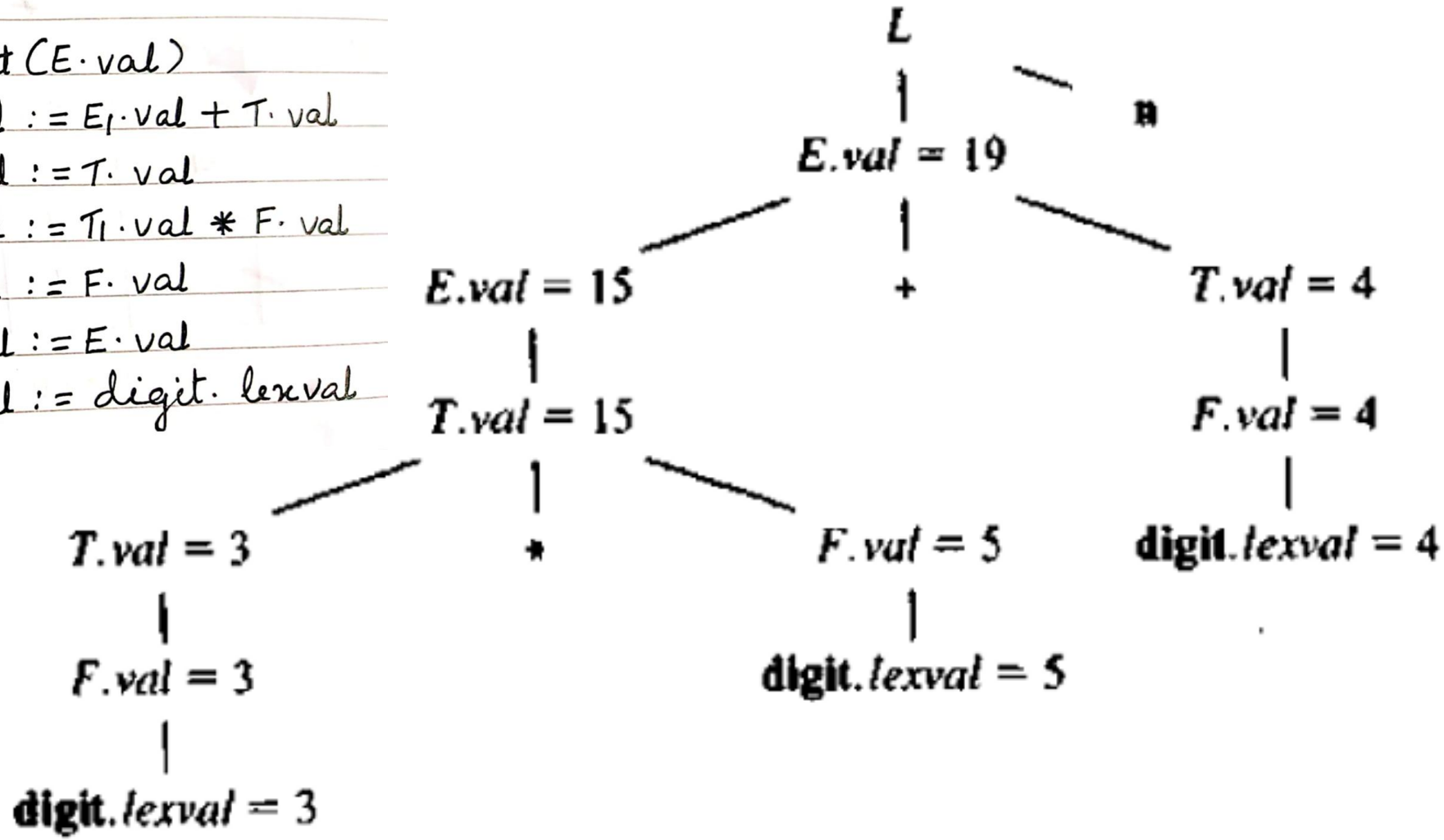
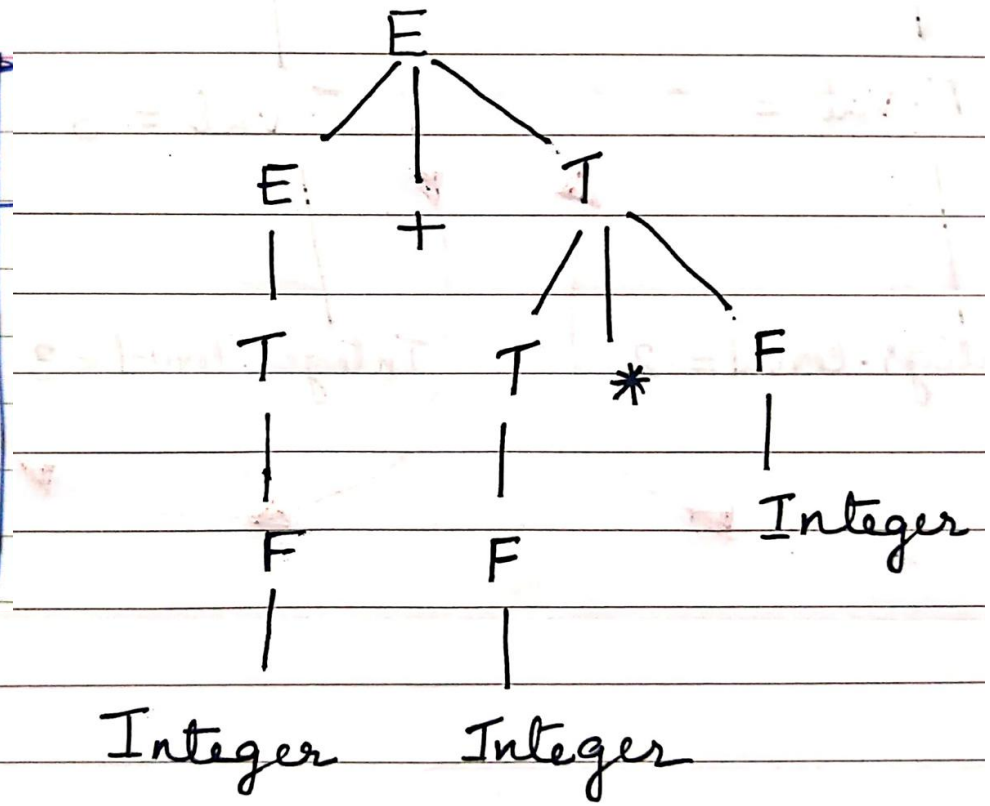


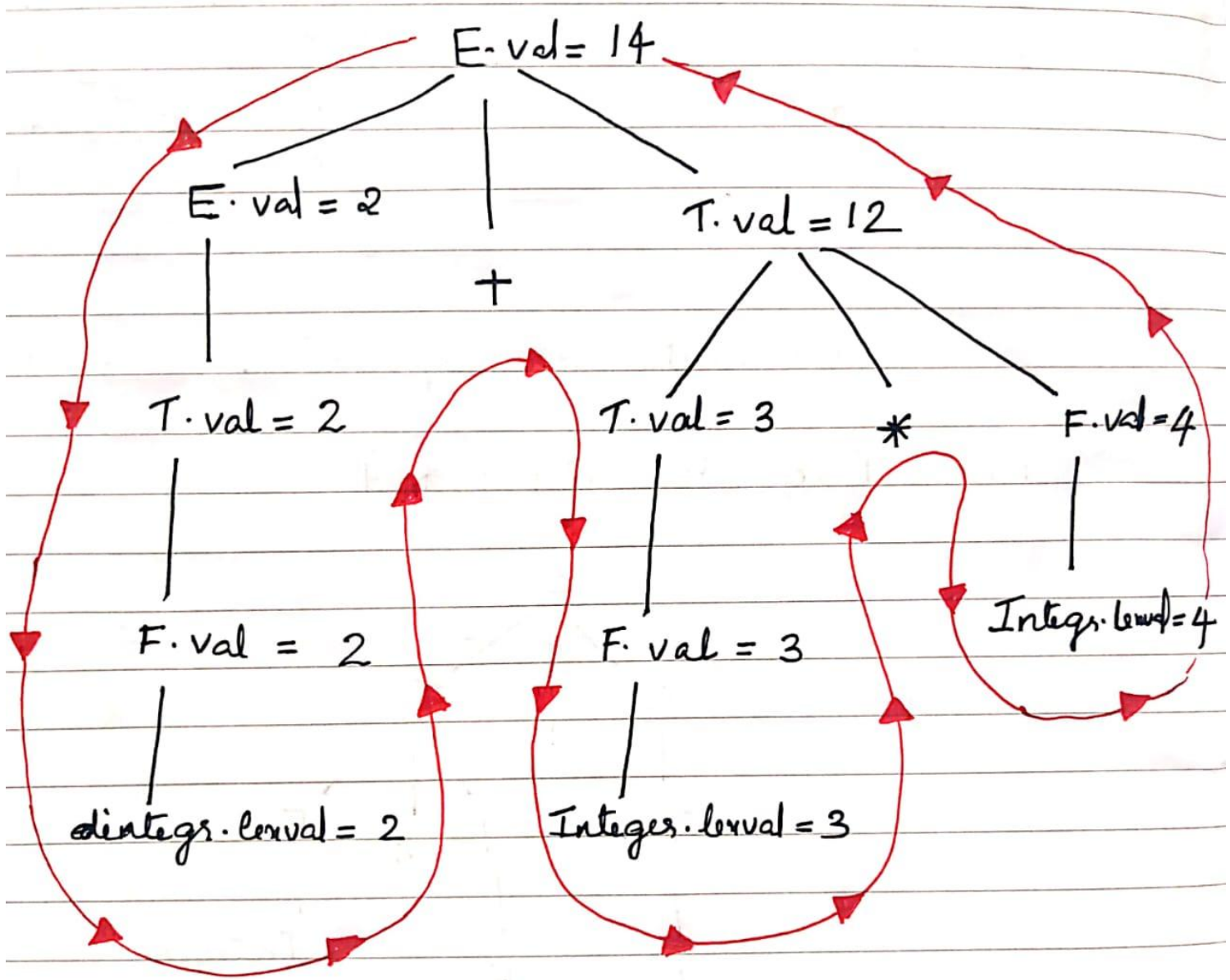
Fig. 5.3. Annotated parse tree for 3*5+4 n.

Parse Tree for 2 + 3 * 4

Grammar	Semantic Rules
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow Integer$	$F.val = Integer.lexval$



Annotated Parse Tree of 2 + 3 * 4

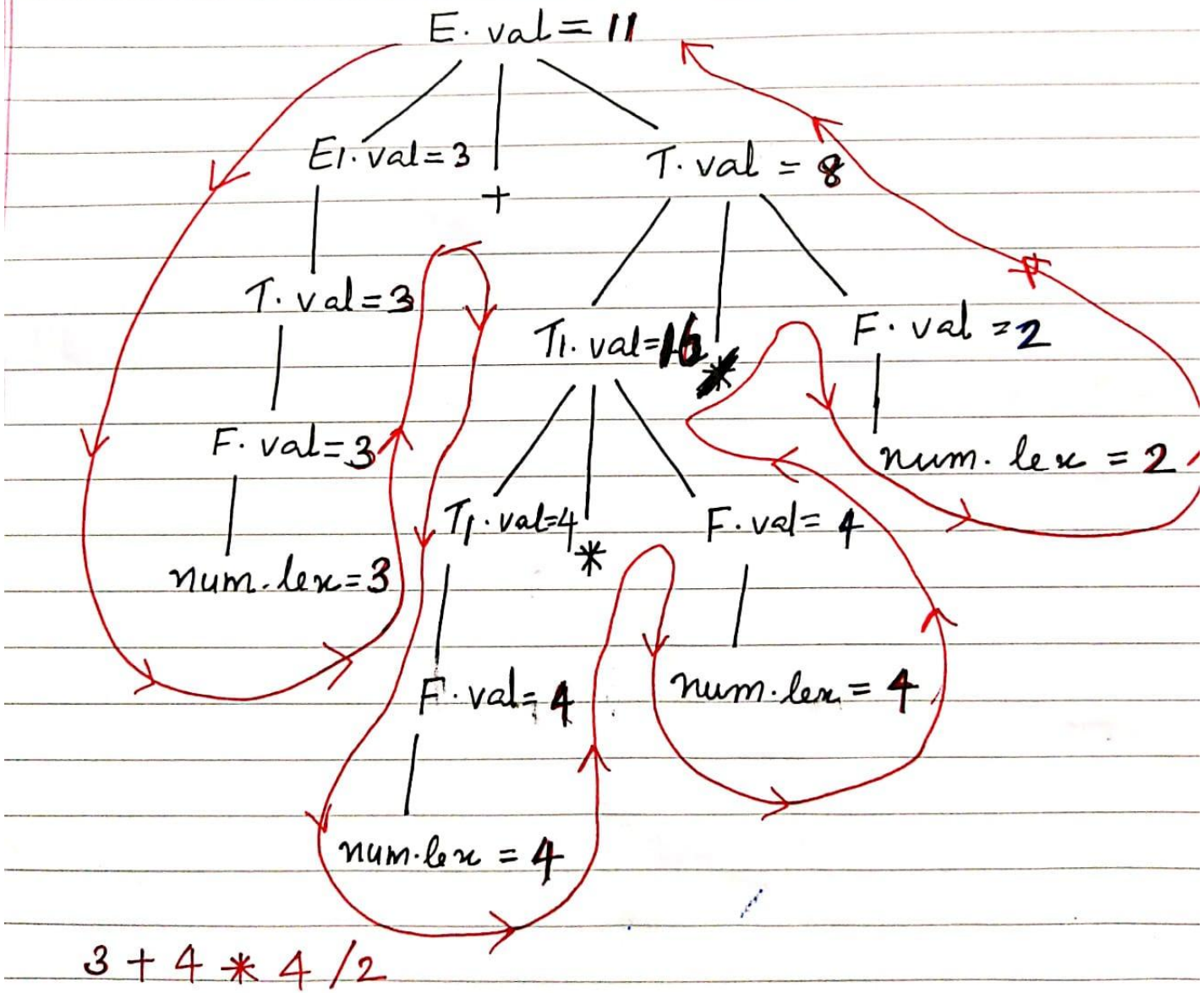


Example :

E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → T1 / F	T.val = T1.val / F.val
T → F	T.val = F.val
F → num	F.val = num.lex

Expression to be evaluated is **3+4*4/2**

Annotated Parse Tree



3 + 4 * 4 / 2

L-attributed definition

- In compiler design, **L-attributed definitions** are a specific type of **Syntax-Directed Definition (SDD)** that ensure the semantic rules can be evaluated in a **left-to-right** order during the traversal of the parse tree.
- It contains both synthesized and inherited attributes.
- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, it is called as L-attributed SDT.

Example:

Example 5.3. A declaration generated by the nonterminal D in the syntax-directed definition in Fig. 5.4 consists of the keyword **int** or **real**, followed by a list of identifiers. The nonterminal T has a synthesized attribute *type*, whose value is determined by the keyword in the declaration. The semantic rule $L.in := T.type$, associated with production $D \rightarrow TL$, sets inherited attribute $L.in$ to the type in the declaration. The rules then pass this type down the parse tree using the inherited attribute $L.in$. Rules associated with the productions for L call procedure *addtype* to add the type of each identifier to its entry in the symbol table (pointed to by attribute *entry*).

PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := integer$
$T \rightarrow$ **real**	$T.type := real$
$L \rightarrow L_1$, **id**	$L_1.in := L.in$ *addtype*(**id**.*entry*, $L.in$)
$L \rightarrow$ **id**	*addtype*(**id**.*entry*, $L.in$)

Syntax-directed definition with inherited attribute $L.in$

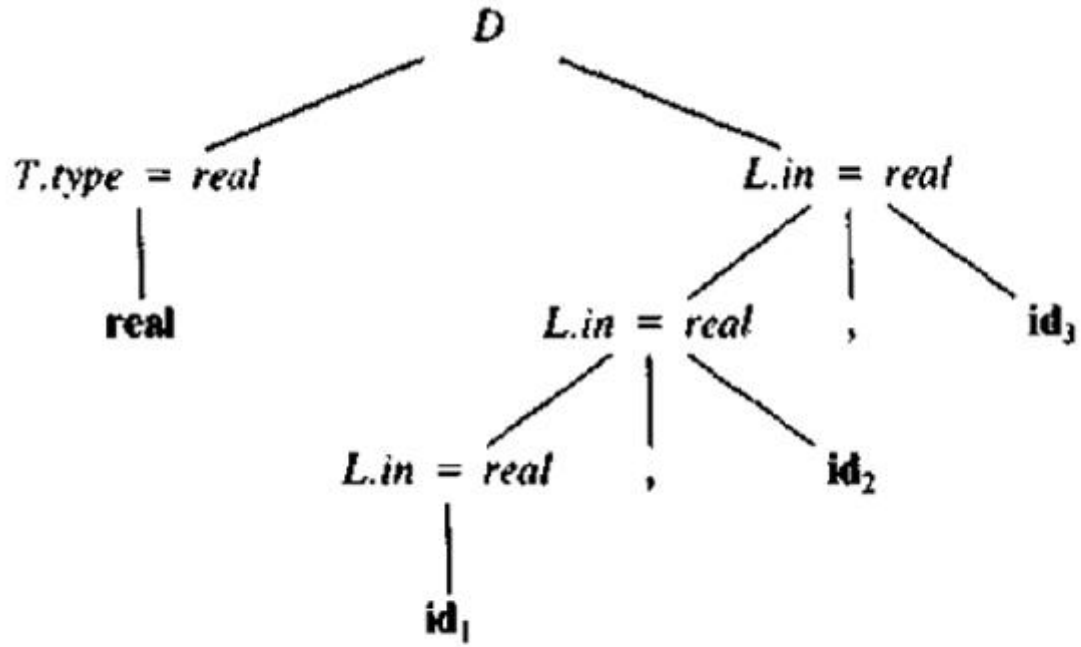


Fig. 5.5. Parse tree with inherited attribute *in* at each node labeled *L*.

Input : real id1, id2, id3

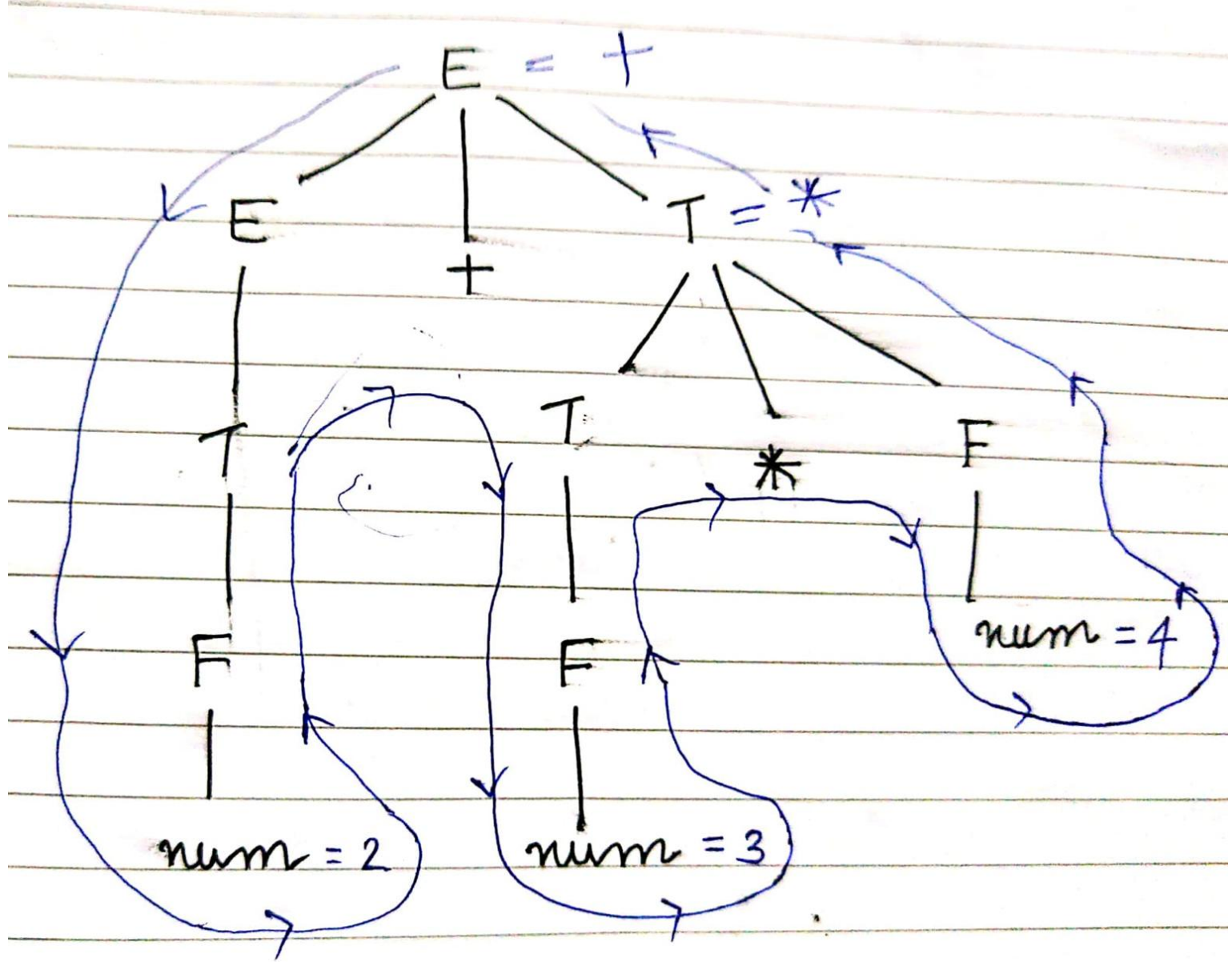
Syntax Directed Translation Schemes (SDT'S)

- Syntax Directed Translation (SDT) is a method used in compiler design where translation is done by augmenting the grammar with semantic actions.
- These actions help in converting the source program into intermediate representations or target code
- **Embeds semantic actions directly into the grammar and these actions execute during parsing.**

Example:

Infix to Postfix Translation

Grammar	Semantic Action
$E \rightarrow E+T$	{ print ("+") }
$E \rightarrow T$	{ }
$T \rightarrow T*F$	{ print ("*") }
$T \rightarrow F$	{ }
$F \rightarrow num$	{ print (num.lexvalue) }



2 3 4 * +

Translate the infix expression 2+3*4 in to its postfix

Dependency Graph

- Semantic rules set up dependencies between attributes that will be represented by a graph.
- A dependency graph is used to represent the flow of information among the attributes in a parse tree.
- In a parse tree, a dependency graph basically helps to determine the evaluation order for the attributes.
- The main aim of the dependency graphs is to help the compiler to check for various types of dependencies between statements in order to prevent them from being executed in the incorrect sequence, i.e. in a way that affects the program's meaning.
- From the dependency graph, we derive an evaluation order for the semantic rules.

- If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
- The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called dependency graph.

PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	$L.in := T.type$
$T \rightarrow$ **int**	$T.type := integer$
$T \rightarrow$ **real**	$T.type := real$
$L \rightarrow L_1$, **id**	$L_1.in := L.in$ *addtype*(**id**.*entry*, $L.in$)
$L \rightarrow$ **id**	*addtype*(**id**.*entry*, $L.in$)

Syntax-directed definition with inherited attribute *L.in*

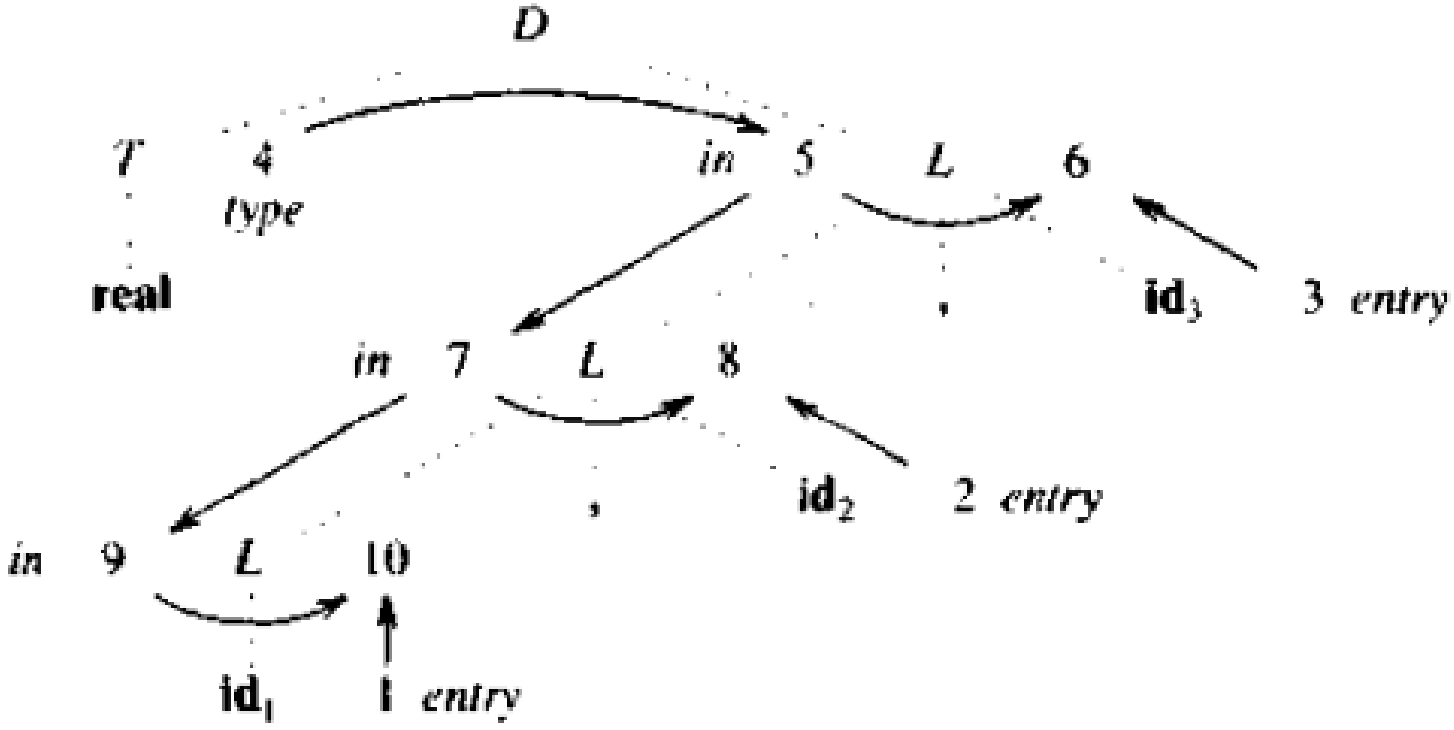


Fig. 5.7. Dependency graph for parse tree of Fig. 5.5.

Constructing syntax tree for expression

- The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.
- We construct subtrees for the subexpressions by creating a node for each operator and operand.
- **Each node in a syntax tree can be implemented as a record with several fields.**
- **In the node for an operator, one field identifies the operator and the remaining fields contains pointers to the node for the operands.**

- **The following functions are used to create the nodes of the syntax tree for expressions with binary operators.**
- Each function returns a pointer to a newly created node.

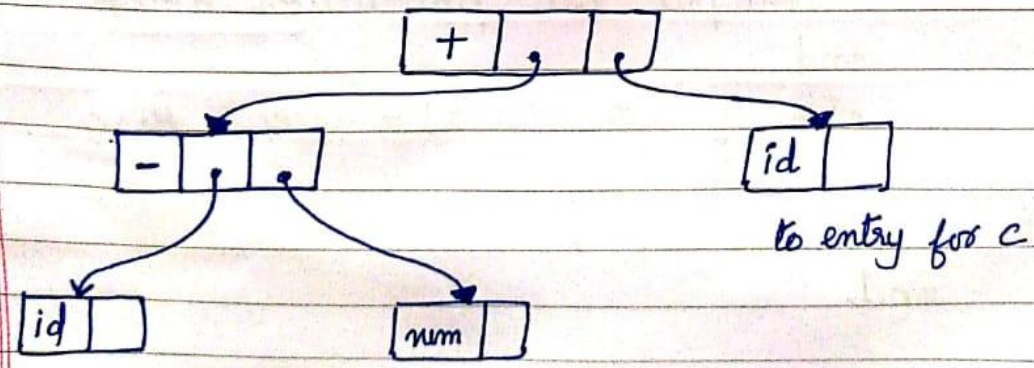
1. *mknode*(*op*, *left*, *right*) creates an operator node with label *op* and two fields containing pointers to *left* and *right*.
2. *mkleaf* (**id**, *entry*) creates an identifier node with label **id** and a field containing *entry*, a pointer to the symbol-table entry for the identifier.
3. *mkleaf* (**num**, *val*) creates a number node with label **num** and a field containing *val*, the value of the number.

Construct syntax tree for the expression
a - 4 + c

Functions

P_1 = mkleaf (id, entry a);
P_2 = mkleaf (num, 4);
P_3 = mknode ('-', P_1, P_2);
P_4 = mkleaf (id, entry c);
P_5 = mknode ('+', P_3, P_4);

- The tree is constructed bottom up.



Syntax tree for a - 4 + c

Figure 5.9 contains an S-attributed definition for constructing a syntax tree for an expression containing the operators + and −. It uses the underlying productions of the grammar to schedule the calls of the functions *mknode* and *mkleaf* to construct the tree. The synthesized attribute *nptr* for E and T keeps track of the pointers returned by the function calls.

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 + T$	$E.nptr := mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow$ **id**	$T.nptr := mkleaf$ (**id**, **id**.*entry*)
$T \rightarrow$ **num**	$T.nptr := mkleaf$ (**num**, **num**.*val*)

Fig. 5.9. Syntax-directed definition for constructing a syntax tree for an expression.

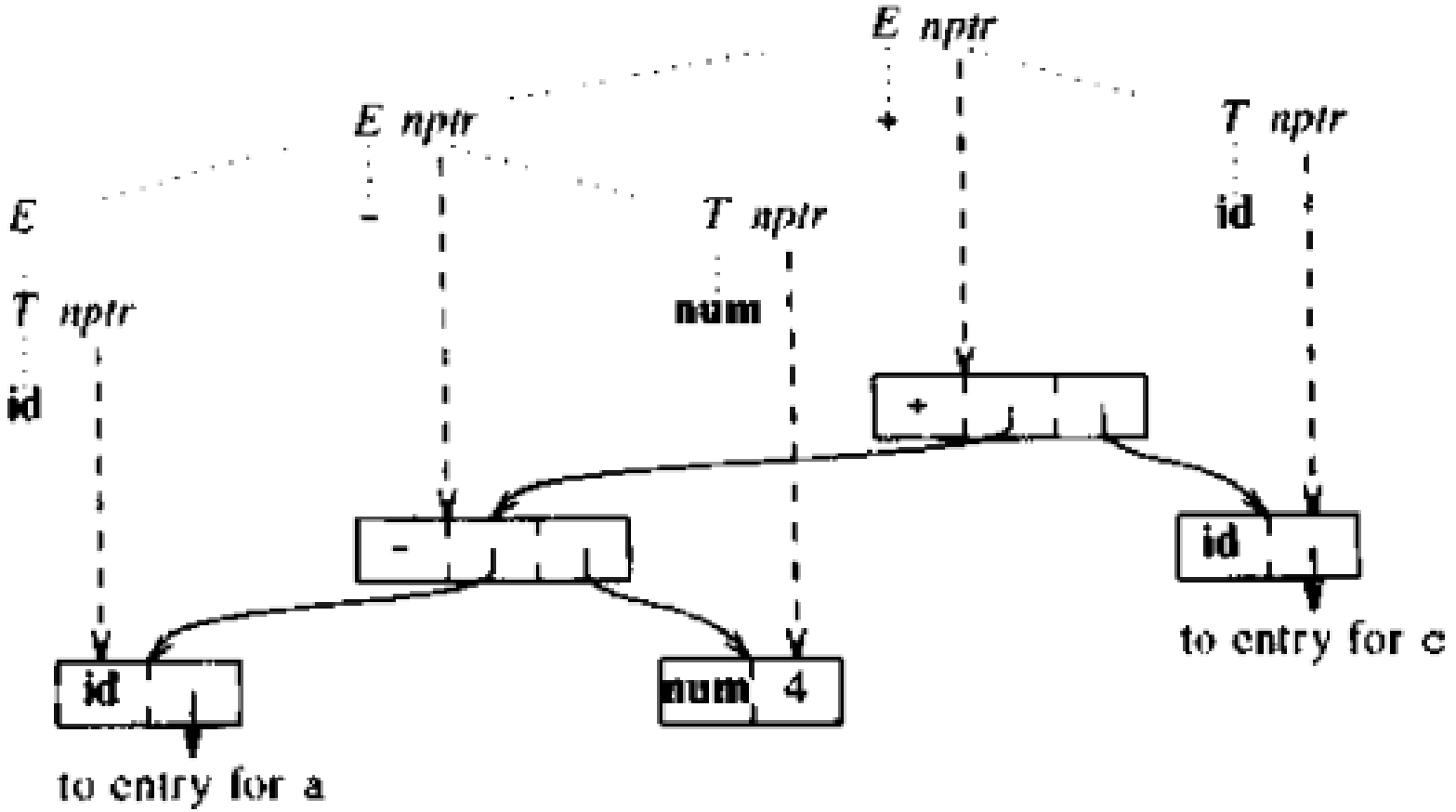


Fig. 5.10. Construction of a syntax-tree for a-4+c.

Directed Acyclic Graph for Expressions

- A DAG is a data structure that represents expressions without redundancy.
- It helps eliminate common subexpressions and detect reuse of values.
- It is used for intermediate representation (IR) and code optimization in compilers.
- **Like a syntax tree, a dag has node for every subexpression of the expression: an interior node represents an operator and its children represents its operands.**

How DAG is constructed?

1. **Step 1:Identify the subexpression**: We analyze the given expression in terms of **operator's precedence and associativity**
2. **Step 2: DAG Construction:**
    - Nodes represent operators and operands.
    - Edges show dependencies between computations.
    - Common subexpressions are merged into a single node.

Figure 5.11 contains a dag for the expression

```
a + a * ( b - c ) + ( b - c ) * d
```

The leaf for a has two parents because a is common to the two subexpressions a and a * (b - c). Likewise, both occurrences of the common subexpression b-c are represented by the same node, which also has two parents.

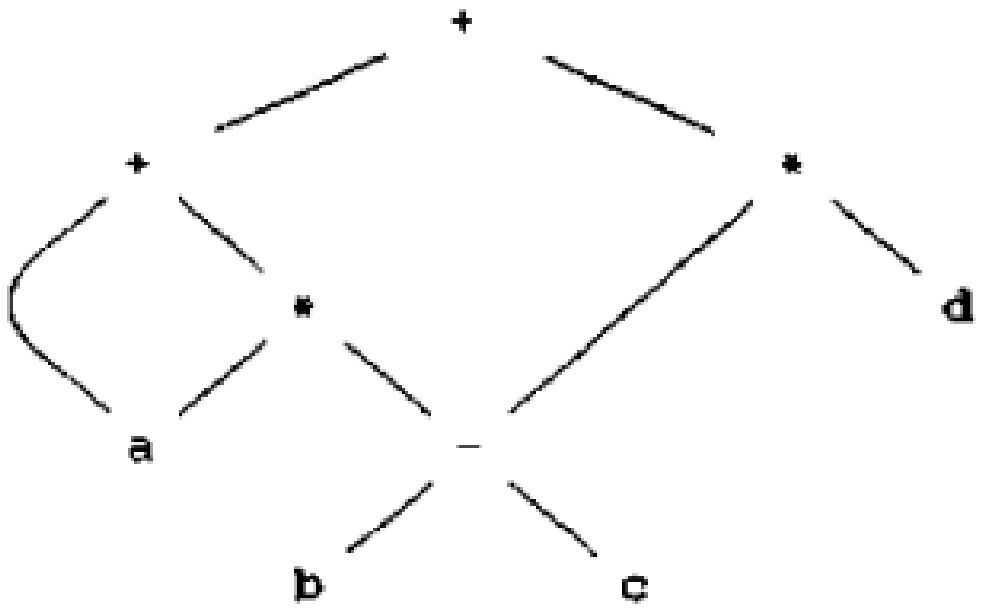


Fig. 5.11. Dag for the expression a+a*(b-c)+(b-c)*d.

Sub expressions are:

1. (b-c)
2. a*(b-c)
3. (b-c)*d
4. a+a*(b-c)

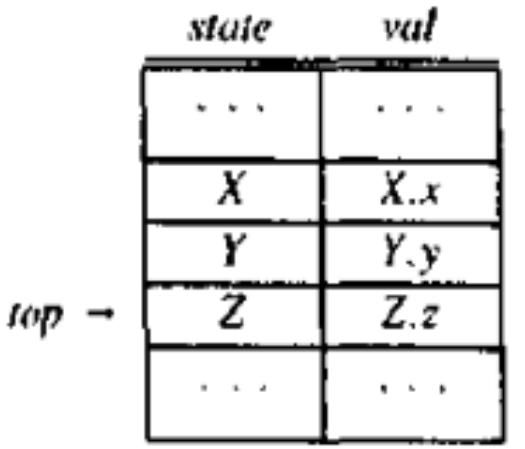
BOTTOM UP EVALUATION OF S-ATTRIBUTED DEFINITION

- **Explains how synthesized attributes are stored and evaluated in a bottom-up parser** (such as an **LR parser**) when dealing with **S-attributed definitions**.
- S-attributed definitions only have synthesized attributes, which means that attributes are computed from child nodes and passed up in the parse tree.
- **Synthesized attributes can be evaluated a bottom-up parser as the input is being parsed**.

- The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack.
- **Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.**
- A translator for an S-attributed definition can often be implemented with help of an LR parser generator.

Synthesized Attributes on the Parser Stack

- **Use extra fields in the parser stack to hold the values of synthesized attributes.**
- **Stack is implemented by a pair of arrays state and val**
- Each entry is a pointer (or index) to an LR(1) parsing table.
- If the ith state symbol is A, then val[i] will hold the value of the attribute associated with the parse tree node corresponding to this.



Parser stack with field for synthesized attributes

- **Before XYZ is reduced to A, the value of the attribute Z.z is in val[top], that of Y.y in val[top-1] and that of X.x in val[top-2].**
- If a symbol has **no attribute, then the entry in the val array is undefined.**
- After the reduction, top is decremented by 2, the state covering A is put in state[top] and value of synthesized attribute A.a is put in val[top].

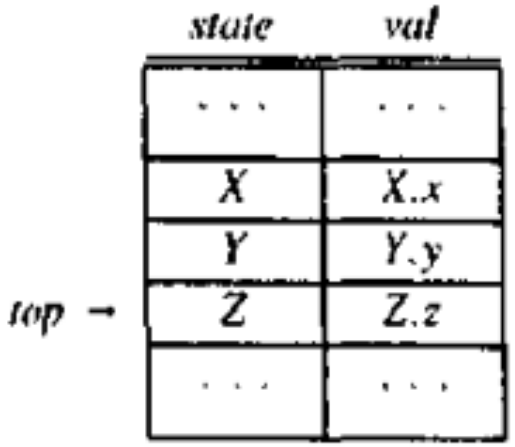


Fig. 5.15. Parser stack with a field for synthesized attributes.

- **The variable ntop is set to the new top of the stack.**
- After a reduction is done top is set to ntop.
- **When a reduction A→α is done with | α |=r , then ntop=top-r+1.**
- During a shift action both the token and its value are pushed into the stack

INPUT	*state*	*val*	PRODUCTION USED
3*5+4 n	_	_	
*5+4 n	3	3	
*5+4 n	F	3	F → digit
*5+4 n	T	3	T → F
5+4 n	T *	3 _	
+4 n	T * 5	3 _ 5	
+4 n	T * F	3 _ 5	F → digit
+4 n	T	15	T → T * F
+4 n	E	15	E → T
4 n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	F → digit
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
	L	19	L → E n

Fig. 5.17. Moves made by translator on input 3*5+4 n.

PRODUCTION	CODE FRAGMENT
$L \rightarrow E$ n	*print* (*val*[*top*])
$E \rightarrow E_1 + T$	$val[ntop] := val[top-2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntop] := val[top-2] \times val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntop] := val[top-1]$
$F \rightarrow$ digit	

Fig. 5.16. Implementation of a desk calculator with an LR parser.

Production	Semantic Rules
$L \rightarrow En$	print(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	F.val := digit.lexval