Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Precedence and Associativity Declarations

- *Use the more natural (ambiguous) grammar*

In CD, operator precedence and associativity are crucial for determining the order in which operations in an expression are evaluated.

Ex: E → E + E | E * E | (E) | id

Ambiguity: Does a + b * c parse as (a + b) * c or a + (b * c)?

- To fix ambiguity, one approach is to **rewrite the grammar** to enforce precedence (priority):

E → E + T | T

T → T * F | F

F → (E) | id

This works, but the grammar becomes more complex and less natural.

Contd.,

Alternative Approach (More Natural)

- Keep the **natural grammar** (even though it's ambiguous).
- Use *precedence and associativity declarations* to disambiguate.
- Most tools allow precedence and associativity declarations to disambiguate grammars

```
E : E '+' E
  | E '*' E
  | '(' E ')'
  | id
  ;

%left '+'
%left '*'
%%
```

Here:

- %left '+' means + is **left associative**.
- %left '*' means * is **left associative** and (because it appears later) it has **higher precedence** than +.

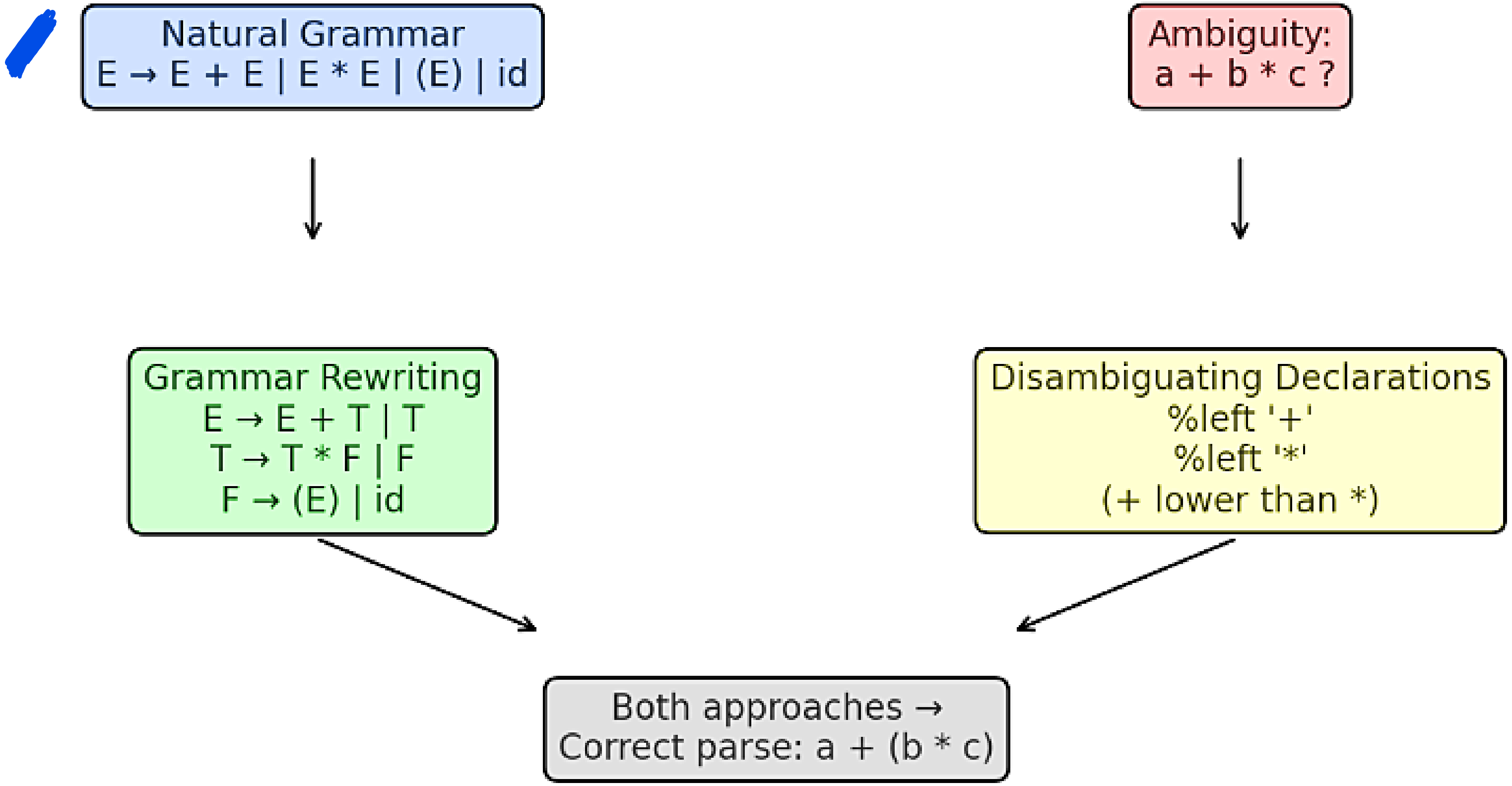


Diagram showing **two approaches to resolve ambiguity** in grammar:

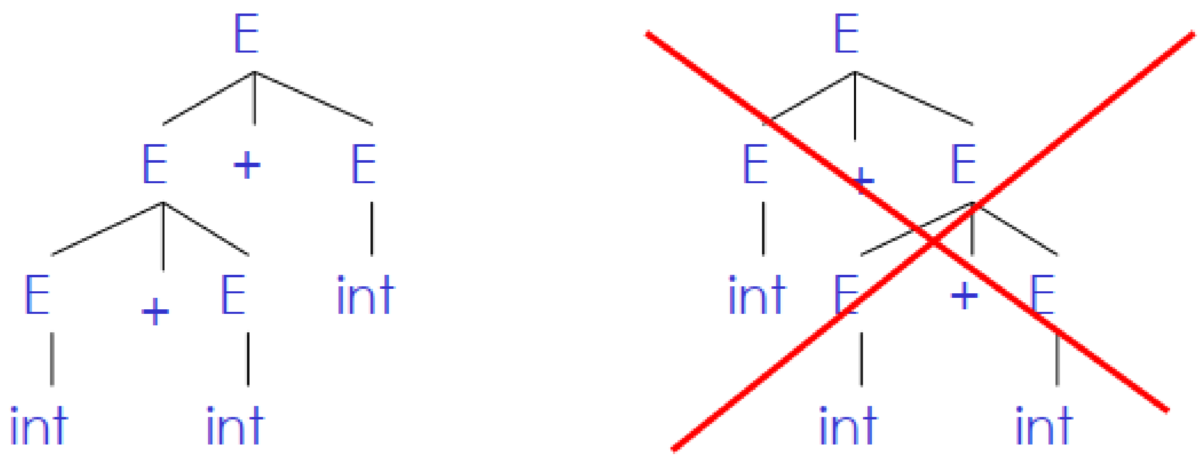
- **Grammar Rewriting** (structuring E, T, F)
- **Disambiguating Declarations** (%left, %right, precedence)

Both lead to the correct parse: **a + (b * c)**

Associativity Declarations

Associativity defines the evaluation order for operators with the same precedence.

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int

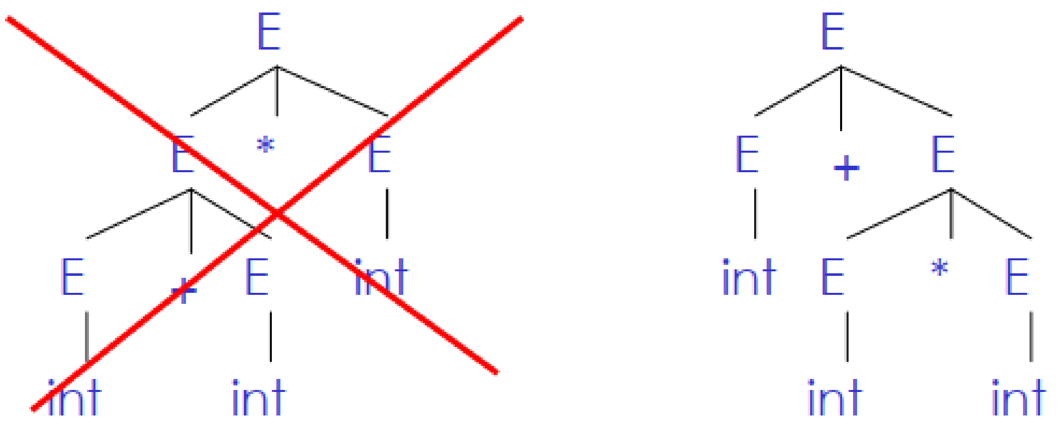


- Left associativity declaration: %left +

Precedence Declarations

Precedence dictates which operator is evaluated first when multiple operators with different priorities are present.

- Consider the grammar E → E + E | E * E | int
  - And the string int + int * int



- Precedence declarations: %left +
  %left *

1) 5 - 3 - 1

Ambiguity: is it (5 - 3) - 1 or 5 - (3 - 1)?

2) 2 + 3 * 4

Ambiguity: (+ (*)) vs (* (+))

3) a ^ b ^ c (exponentiation)

Ambiguity: (a ^ b) ^ c vs a ^ (b ^ c)

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by ...
Lexical	... $ ...	Lexer
Syntax	... x *% ...	Parser
Semantic	... int x; y = x(3); ...	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- When a compiler encounters an error in the source program, it must **detect**, **report**, and, if possible, **recover** from it without stopping compilation completely.
- Error handler should
  - Report errors accurately and clearly
    - Error messages should be informative (line number, nature of error).
    - Example: instead of just *"Syntax error"*, say *"Missing semicolon before '}' at line 12"*.
  - Recover from an error quickly
    - The compiler should attempt to skip or fix the error and continue parsing.
    - One mistake causing many false error reports.
  - Not slow down compilation of valid code
    - Error handling must not add overhead when code is correct.
    - Normal compilation speed should be preserved.
- ***Good error handling is not easy to achieve***

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction
    - Perform local corrections (insert, delete, replace tokens).
    - Example: insert missing semicolon automatically.
    - Theoretical method: change program minimally to make it syntactically correct.
    - Too expensive, rarely used in practice.
- Not all are supported by all parser generators

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found **(ex, ; or }).**
  - Continue from there
- Such tokens are called synchronizing tokens
  - Typically, the statement or expression terminators

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

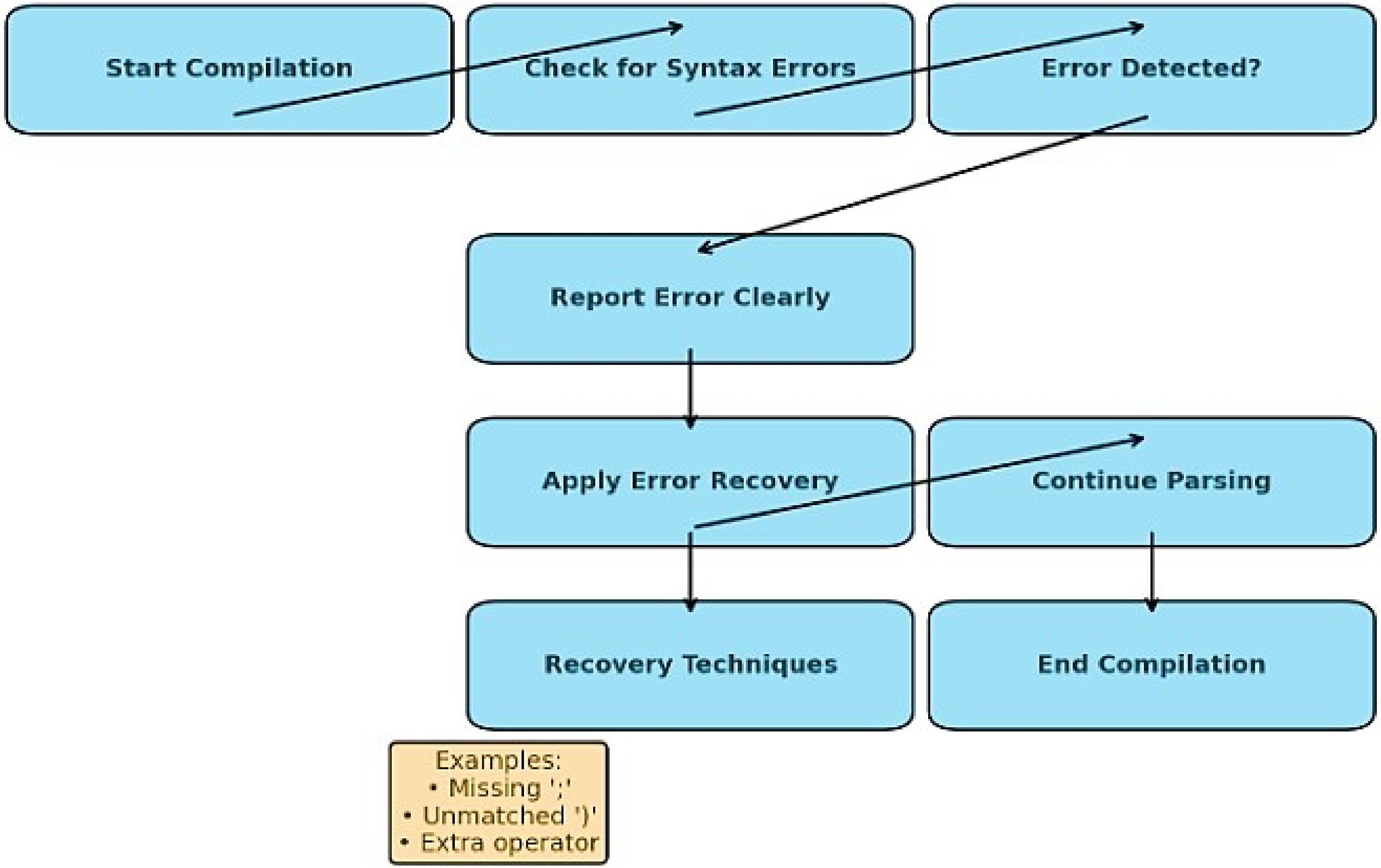
- (ML)-Yacc: use the special terminal error to describe how much input to skip

  $E \rightarrow$ int | E + E | (E) | error int | (error)

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write **5 x** instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$
- Disadvantage
  - Complicates the grammar

Flowchart of syntax error handling



Syntax Error Recovery: Past and Present

- Past
  - **Slow recompilation cycle** – Rebuilding the entire program used to take hours, sometimes even a whole day.
  - Because of this, the compiler tried to **detect as many errors as possible in one run**, so programmers wouldn't have to wait another day.
  - Researchers put huge effort into **complex error recovery techniques** (error productions, sophisticated heuristics, etc.).

- Present
  - **Fast recompilation** – Now, with modern hardware and incremental compilation, rebuilding takes seconds.
  - Programmers usually **fix one error at a time**, recompile, and repeat.
  - Therefore, we don't need **very complex error recovery** strategies anymore.
  - **Panic-mode recovery** (simple, discard tokens until a synchronizing point) is usually **good enough**

Compiler

Abstract Syntax Trees

&

Top-Down Parsing

Review of Parsing

- Given a language L(G), a parser consumes a sequence of tokens **s** and produces a parse tree
- Issues:
  - – How do we recognize that s ∈ L(G) ?
  - – A parse tree of s describes how s ∈ L(G)
  - – Ambiguity: more than one parse tree (possible interpretation) for some string s
  - – Error: no parse tree for some string s
  - – How do we construct the parse tree?

Abstract Syntax Trees

- So far, a parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

- Consider the grammar

  $E \rightarrow \text{int} \mid (E) \mid E + E$

- And the string

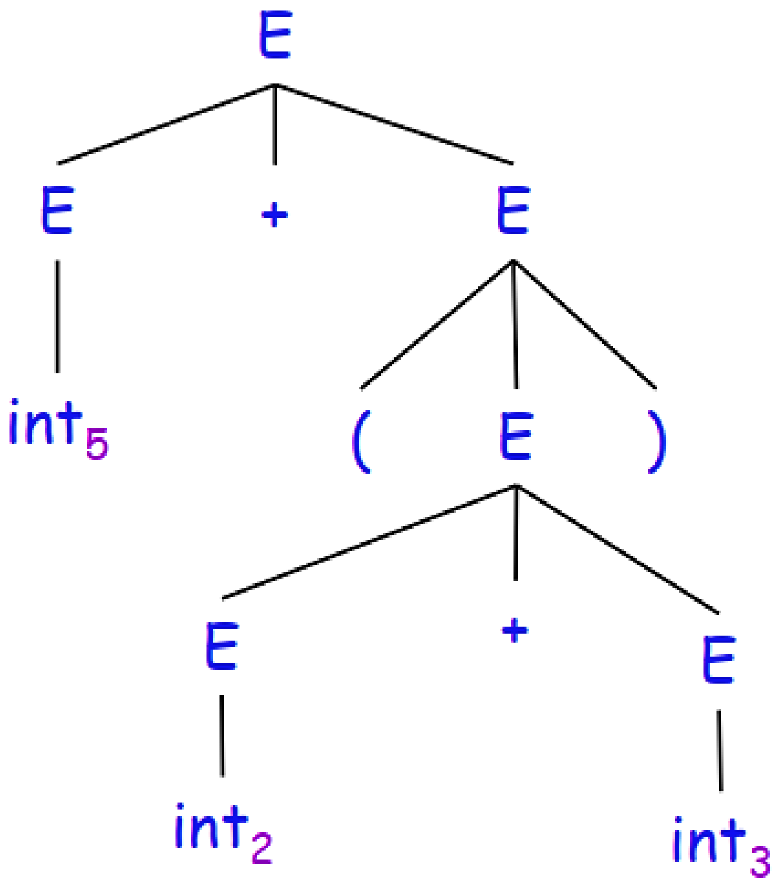
  $5 + (2 + 3)$

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

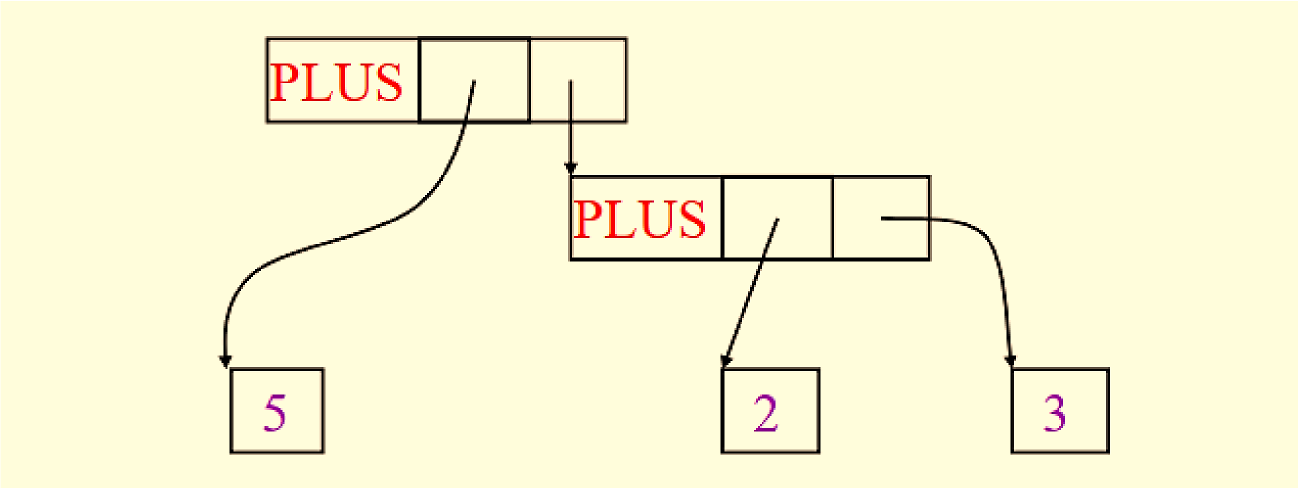
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Captures the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Example of Abstract Syntax Tree



- Abstracts from the concrete syntax a more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we'll use to construct ASTs
- Each grammar symbol may have attributes
  - An attribute is a property of a programming language construct
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an **action**
  - Written as: X → Y1 … Yn   **{ action }**
  - That can refer to or compute symbol attributes

Semantic Actions: An Example

- Consider the grammar

$$E \to \text{int} \mid E + E \mid (E)$$

- For each symbol X define an attribute X.val
  - For terminals, val is the associated lexeme
  - For non-terminals, val is the expression's value (which is computed from values of subexpressions)
- We annotate the grammar with actions:

$$
\begin{array}{lll}
E \to \text{int} & & \{ E.val = \text{int}.val \} \\
\quad \mid E_1 + E_2 & & \{ E.val = E_1.val + E_2.val \} \\
\quad \mid (E_1) & & \{ E.val = E_1.val \}
\end{array}
$$

Semantic Actions: An Example (Cont.)

- String: 5 + (2 + 3)
- Tokens: int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int_5$	$E_1.val = int_5.val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int_2$	$E_4.val = int_2.val = 2$
$E_5 \rightarrow int_3$	$E_5.val = int_3.val = 3$

Semantic Actions: Dependencies

- Semantic actions specify a system of equations
  - Order of executing the actions is not specified

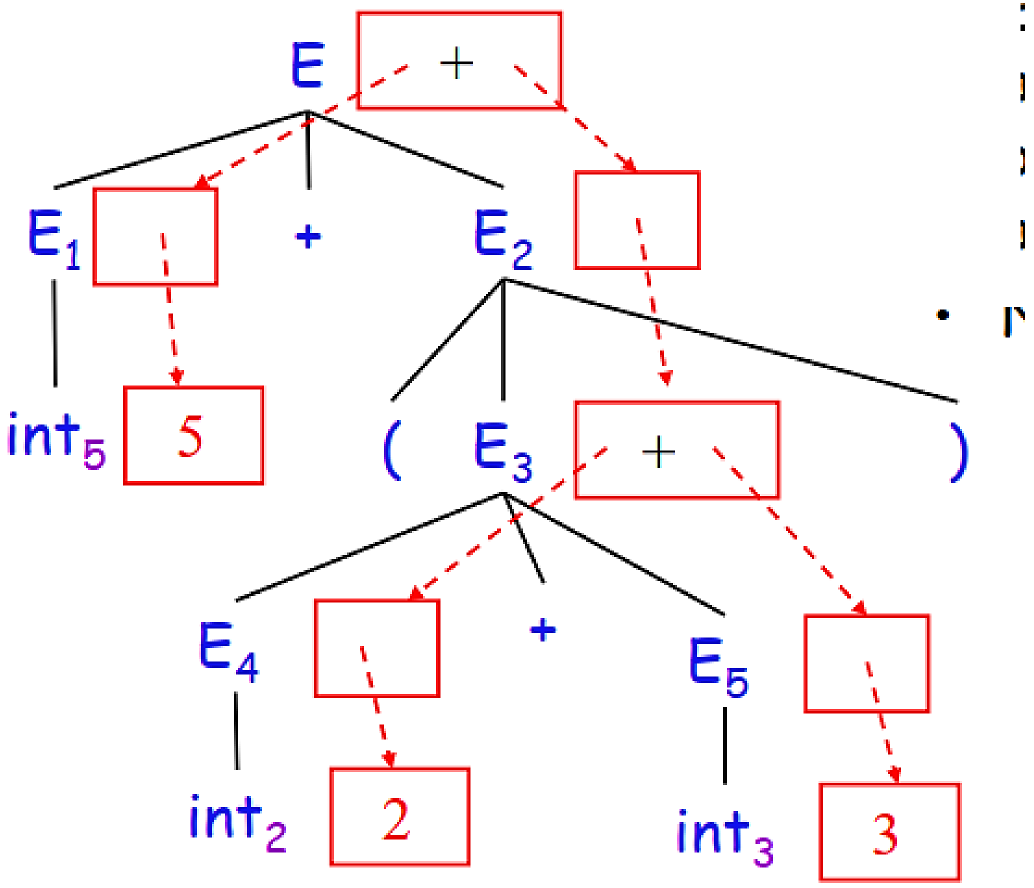
- Example:

  $E_3.val = E_4.val + E_5.val$

  - Must compute $E_4.val$ and $E_5.val$ before $E_3.val$
  - We say that $E_3.val$ *depends on* $E_4.val$ and $E_5.val$

- The parser must find the order of evaluation

Dependency Graph



Dependency Graph or Directed graph used to represent flow of information and order of evaluation among attributes within a parse tree.

Meaning of Red Boxes and Dashed Arrows:

- **Red boxes** represent the operators and constants.
- Dashed arrows indicate evaluation flow:
  - For example, *from E_1 we go to constant 5.*
  - *From E_3 we go down to its + operator*.

Why parentheses needed? Without parentheses, ambiguity arises in arithmetic grammar. Here, parentheses explicitly force evaluation order: 5+(2+3)=10.

Evaluating Attributes in Compiler

- Attributes = values attached to grammar symbols (like numbers, types, or computed results).
- Dependency graph = shows how each attribute depends on others.
- To evaluate attributes correctly: You must compute an attribute **only after** all the attributes it depends on have been computed.

- An attribute must be computed after all its successors in the dependency graph have been computed

  Means: follow the dependencies like a DAG (Directed Acyclic Graph).

  If E.value = E1.value + E2.value, you must compute E1.value and E2.value first.
- In the previous example attributes can be computed bottom-up

  You compute the values from leaves upward (constants first, then operators).

  Example: Expression = 5 + (2 + 3)
- Such an order exists when there are no cycles

  If the dependency graph has **no cycles**, you can always find an order to evaluate attributes.

  Example: Expression trees are acyclic, so evaluation works fine.

Semantic Actions: Notes (Cont.)

- Synthesized attributes
  - Calculated from attributes of descendants in the parse tree

    $$E \rightarrow E1 + T$$

    $$E.val = E1.val + T.val$$

  - E.val is a synthesized attribute from E1.val + T.val
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called S-attributed grammars

  Means: A grammar is **S-attributed** if **all attributes are synthesized**.

  Advantages:

  - Easy to implement with bottom-up parsers (like LR parsers).
  - No need to look upward or sideways in the tree.

Inherited Attributes

- Another kind of attributes: Calculated from attributes of the parent node(s) and/or siblings in the parse tree

Example: a line calculator

- Each line contains an expression

  $E \rightarrow int \mid E + E$

- Each line is terminated with the = sign

  $L \rightarrow E = \mid + E =$

- In the second form, the value of evaluation of the previous line is used as starting value
- A program is a sequence of lines

  $P \rightarrow \varepsilon \mid P\ L$

Why Inherited Attributes?

- In the rule L → +E=,
- the evaluation of E **depends on the value of the previous line**.
- That means we must **pass information from parent or left siblings down to E**.
- This is not possible with synthesized attributes alone → we need **inherited attributes**.

Attributes for the Line Calculator

- Each E has a synthesized attribute val
  - Calculated as before
- Each L has a synthesized attribute val

$$
\begin{aligned}
L &\to E = && \{\ L.val = E.val\ \} \\
&\mid\ +E = && \{\ L.val = E.val + L.prev\ \}
\end{aligned}
$$

- We need the value of the previous line
- We use an inherited attribute L.prev

Ex: 3+2=
+4=

- First line: E.val = 5, so L.val = 5.
- Second line: L.prev = 5, E.val = 4, so L.val = 5 + 4 = 9.
- Final result: P.val = 9.

- Why do we need L.prev?
- In the case L → +E=, the line's evaluation depends on the **previous line's result**.
- Since that information comes from outside this production, we must pass it in as an **inherited attribute**.

Attributes for the Line Calculator (Cont.)

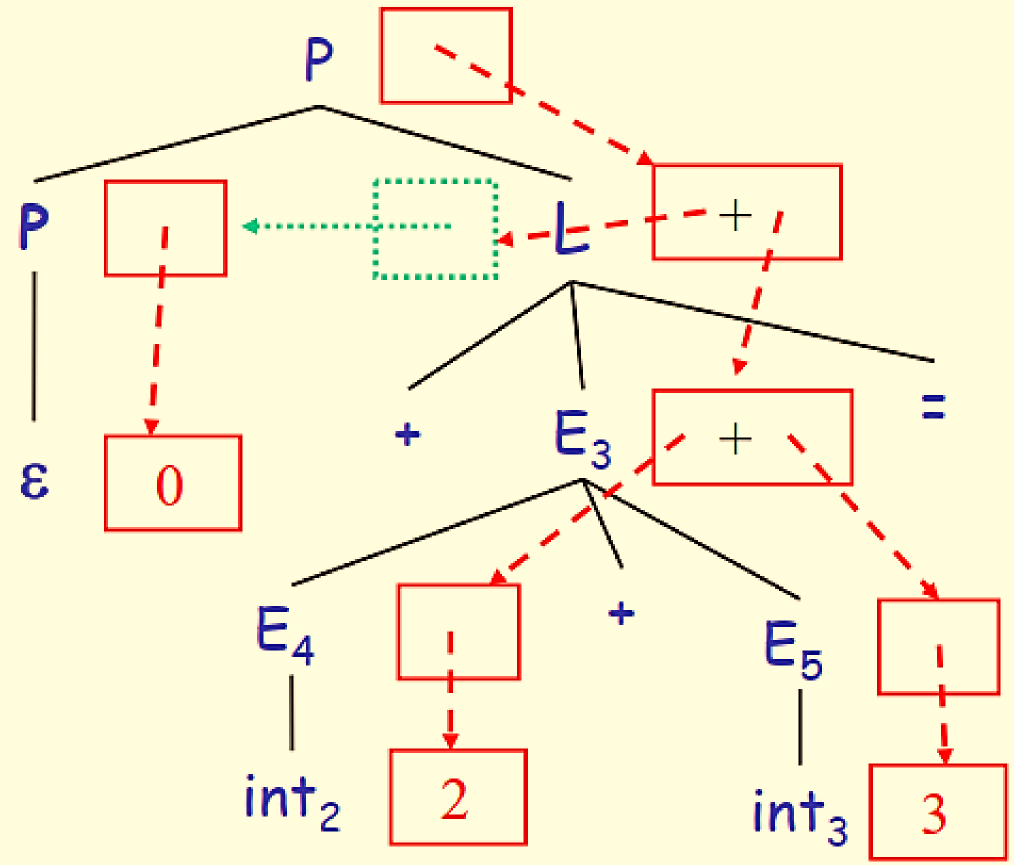
- Each P has a synthesized attribute val
  - The value of its last line

$$
\begin{aligned}
P &\to \varepsilon && \{\ P.val = 0\ \} \\
&\mid\ P_1\ L && \{\ P.val = L.val; \\
& && \quad L.prev = P_1.val\ \}
\end{aligned}
$$

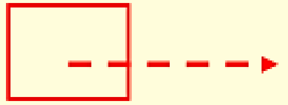
- Each L has an inherited attribute prev
  - L.prev is inherited from sibling P_1.val

- Example …

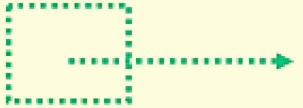
Example of Inherited Attributes



- val synthesized



- prev inherited



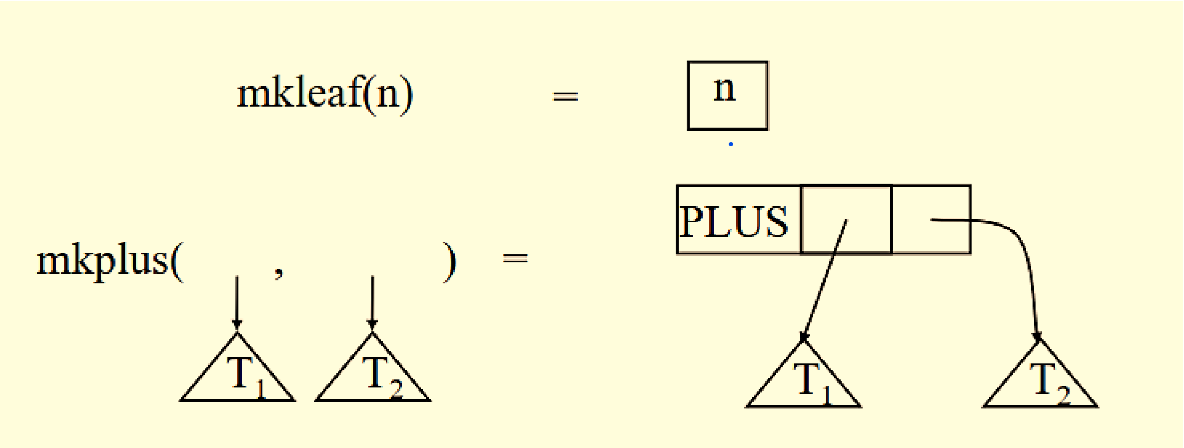
- All can be computed in depth-first order

Semantic Actions: Notes (Cont.)

- Semantic actions can be used to build ASTs
- And many other things as well
  - Also used for type checking, code generation, …
- Process is called syntax-directed translation
  - Substantial generalization over CFGs

Constructing an AST

- We first define the AST data type
- Consider an abstract tree type with two constructors



Constructing a Parse Tree

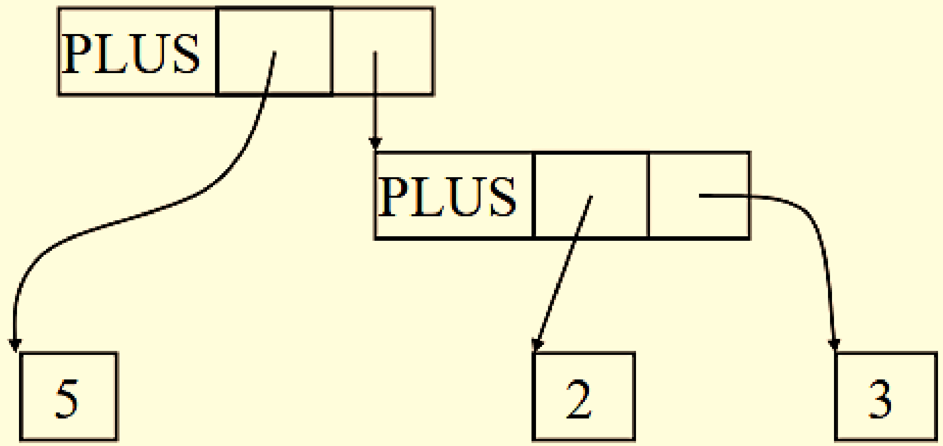
- We define a synthesized attribute **ast**
  - Values of **ast** values are ASTs
  - We assume that int.**lexval** is the value of the integer lexeme
  - Computed using semantic actions

$$
\begin{array}{lll}
E \rightarrow & int & \{\ E.ast = mkleaf(int.lexval)\ \} \\
 \mid & E_1 + E_2 & \{\ E.ast = mkplus(E_1.ast,\ E_2.ast)\ \} \\
 \mid & (\ E_1\ ) & \{\ E.ast = E_1.ast\ \}
\end{array}
$$

Parse Tree Example

- Consider the string int_5 '+' '(' int_2 '+' int_3 ')'
- A bottom-up evaluation of the ast attribute:

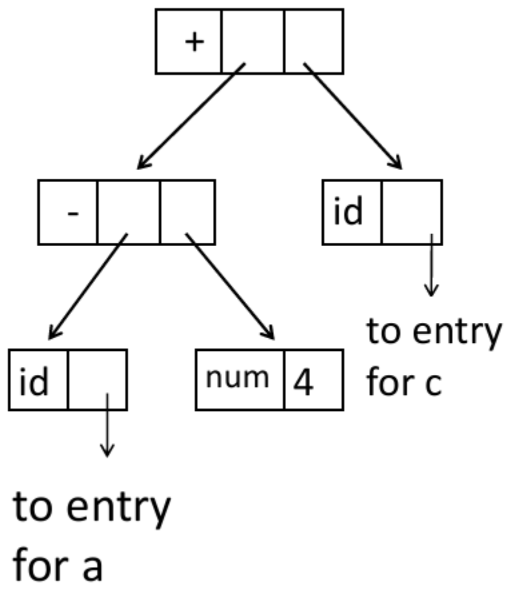
  E.ast = mkplus(mkleaf(5),
  mkplus(mkleaf(2), mkleaf(3))



Constructing Syntax Tree for Expressions

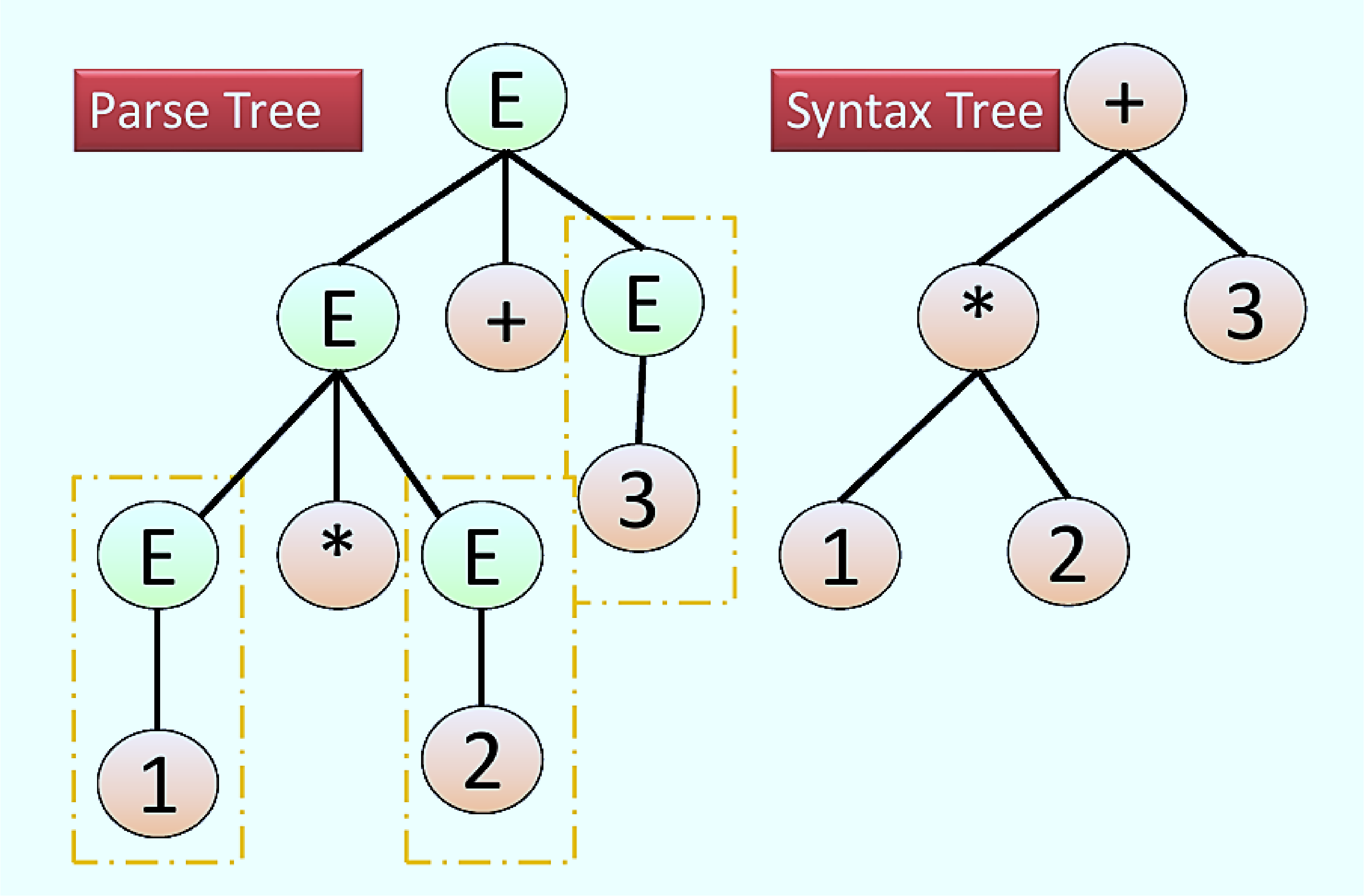
- Example: **a-4+c**

1. p1:=mkleaf(id,entrya);
2. p2:=mkleaf(num,4);
3. p3:=mknode(-,p1,p2)
4. p4:=mkleaf(id,entryc);
5. p5:= mknode(+,p3,p4);



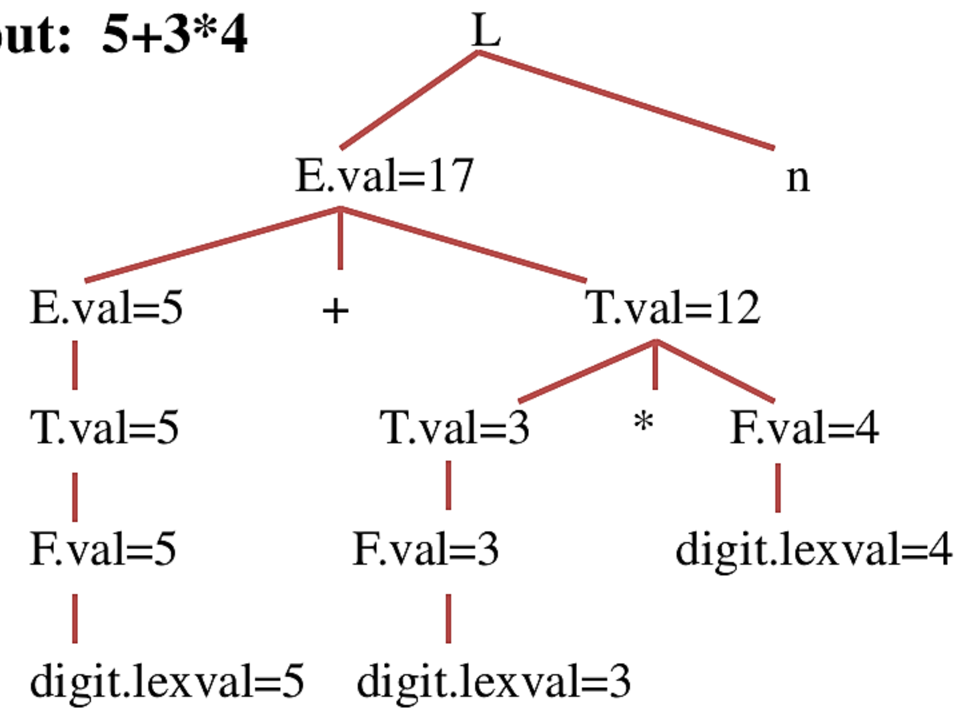
The tree is constructed bottom up.

Parse Tree Vs Syntax Tree : 1*2+3



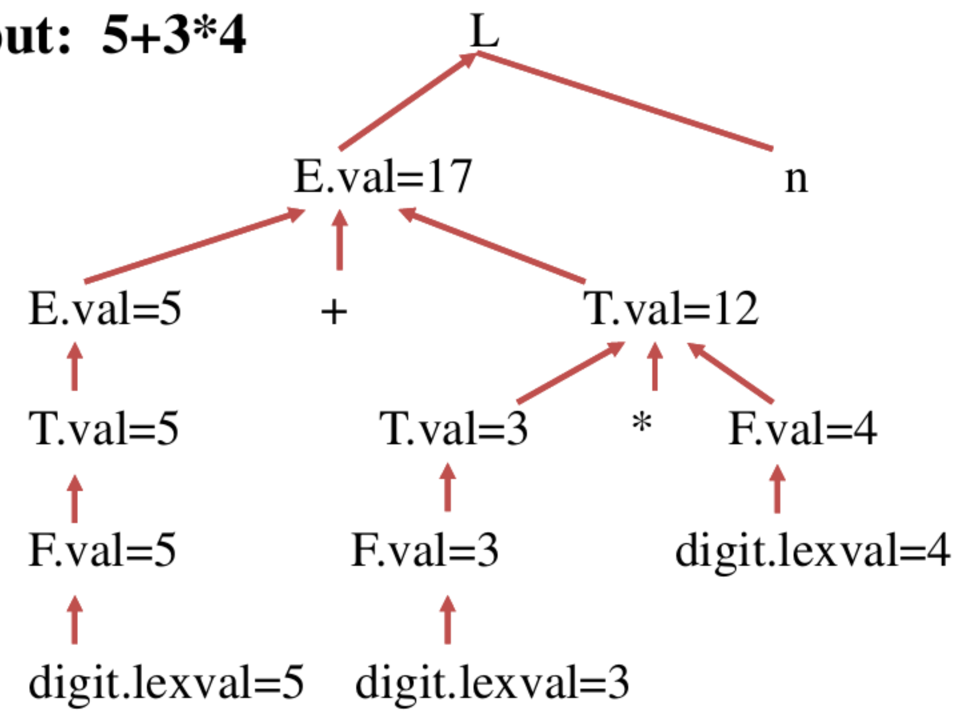
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



Parse Tree Vs Syntax Tree : 1*2+3

Parse Tree

- Parse Tree contain operators & operands at any node of the tree, i.e., either interior node or leaf node.
- Parse contains duplicate or redundant information.
- Parse Tree can be changed to Syntax Tree by the elimination of redundancy, by Compaction

Syntax Tree

- Syntax contains operands at leaf node & operators as interior nodes of Tree.
- ST do not contains duplicate info
- Syntax Tree cannot be changed to Parse Tree

Review of Abstract Syntax Trees

- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
- … and will build a parse tree
- … which we convert to an AST
- … and pass on to the rest of the compiler

- Next part:
  - How do we answer $s \in L(G)$ and build a parse tree?