IT 302 Compiler Design

Semantic and Syntax Directed Translation

The Compiler so far

- Lexical analysis: program is *lexically well-formed*
  - Tokens are legal (e.g. identifiers have valid names, no lost characters, etc.)
  - Detects inputs with illegal tokens
- Parsing: program is *syntactically well-formed*
  - Declarations have corrected structure, expressions are syntactically valid, etc.
  - Detects inputs with ill-formed syntax
- Semantic analysis:
  - Last “front end” compilation phase
  - Catches all remaining errors

Why have a Separate Semantic Analysis?

- Parsing cannot catch some errors
- Some language constructs are not context-free
  - Example: Identifier declaration and use
  - An abstract version of the problem is:

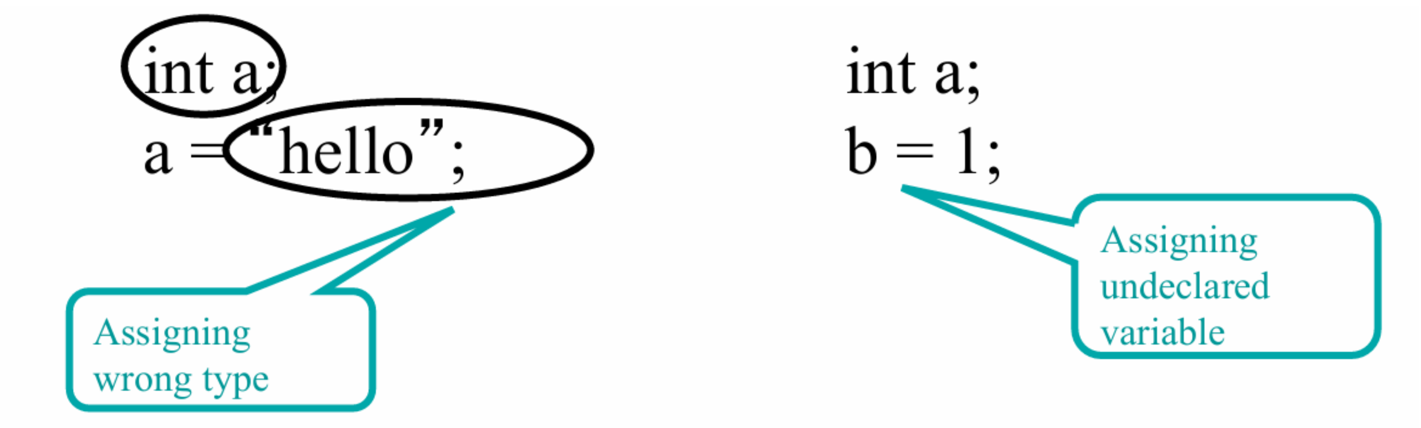
$$L = \{ wcw \mid w \in (a + b)^* \}$$

  - The 1st w represents the identifier’s declaration;
  - the 2nd w represents a use of the identifier
  - C separates the declaration and use parts

abcab ✅ valid (same 'w' before and after 'c')
abbcabb ✅ valid
abcabb ❌ invalid (mismatch)

Contd.,

- Syntactically correct programs may still contain errors
  - Lexical analysis does not distinguish between different variable names (same ID token)
  - Syntax analysis does not correlate variable declaration with variable use, does not keep track of types



Goals of semantic analysis

- Check “correct” use of programming constructs
- Provide information for subsequent phases
- Context-sensitive – beyond context-free grammars
  - Lexical analysis and syntax analysis provide relatively shallow checks of program structure
  - Semantic analysis goes deeper
- Correctness specified by semantic rules
  - Scope rules
  - Type-checking rules
  - Specific rules
- Note: semantic analysis ensures only partial correctness of programs – Runtime checks (pointer dereferencing, array access)

Example of semantic rules

- A variable must be declared before used
- A variable should not be declared multiple times
- A variable should be initialized before used
- Non-void method should contain return statement along all execution paths
- *break/continue* statements allowed only in loops
- *this* keyword cannot be used in static method
- *main* method should have specific signature
- …
- Type rules are important class of semantic rules
  - In an assignment statement, the variable and assigned expression must have the same type
  - In a condition test expression must have a boolean type

What Does Semantic Analysis Do?

- Performs checks of many kinds ...
- Examples:
  - 1. All used identifiers are declared
  - 2. Identifiers declared only once
  - 3. Types
  - 4. Procedures and functions defined only once
  - 5. Procedures and functions used with the right number and type of arguments
  - And others . . .
- The requirements depend on the language

Semantic Analysis

- Compilers examine code to find semantic problems.
  - Easy: undeclared variables, tag matching
  - Difficult: preventing execution errors

- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

Role of Syntax-Directed Translation (SDT)

- To associate actions with productions
- To associate attributes with non-terminals
- To create an implicit (hidden) or explicit (clear) syntax tree
- To perform semantic analysis …
- ……………………
- ……………………
- Essentially, to add life to the skeleton

Example

E → E + T	$$.code = “”;
	strcat($$.code, $1.code);
	strcat($$.code, $3.code);
	strcat($$.code, “+”);

SDD

Attributes

E → E + T	{ printf(“+”); }

SDT

SDTs may be viewed as implementations of SDDs and are important from efficiency perspective.

Productions **Actions**

Syntax Directed Definition

- An SDD is a CFG with attributes and rules.
  - Attributes are associated with grammar symbols.
  - Rules are associated with productions.

- An SDD specifies the semantics of productions.
  - It does not enforce a specific way of achieving the semantics.

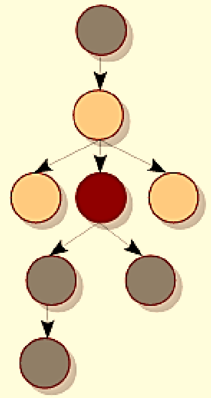
Syntax Directed Translation

- An SDT is done by attaching rules or program fragments to productions.

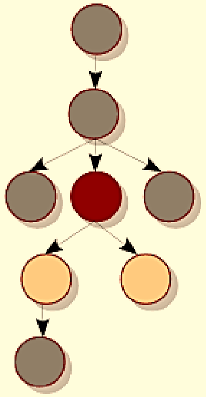
- The order prompted by the syntax analysis produces a translation of the input program.

Attributes

- Inherited
  - In terms of the attributes of the node, its parent and siblings.
  - e.g., int x, y, z; or nested scoping



- Synthesized
  - In terms of the attributes of the node and its children.
  - e.g., a + b * c or most of the constructs from your assignments



SDD for Calculator

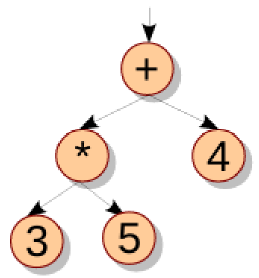
Input string

3 * 5 + 4 $

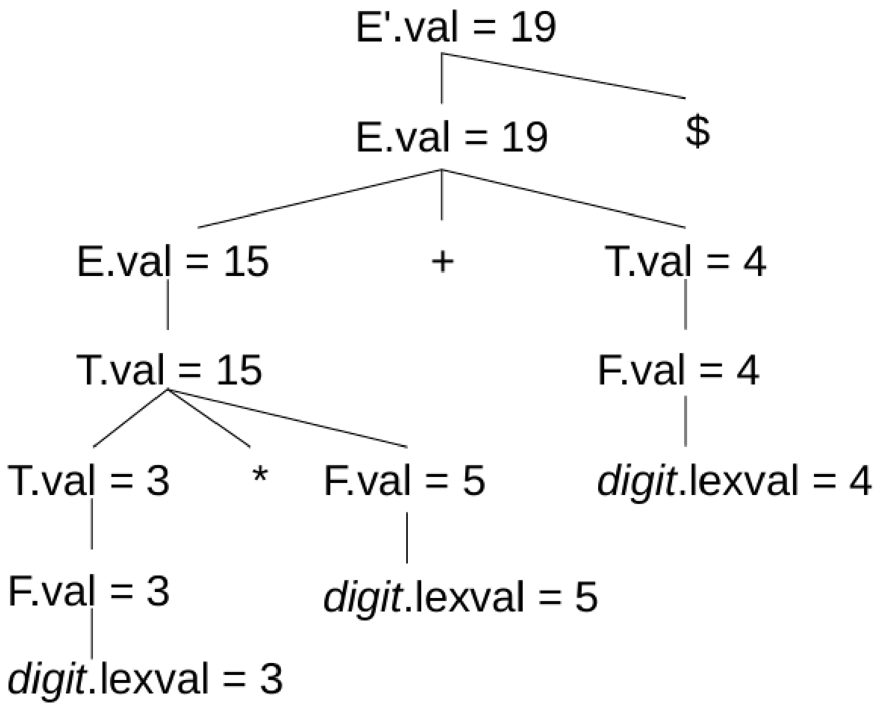
SDD

Sr. No.	Production	Semantic Rules
1	$E' \rightarrow E\ \$$	E'.val = E.val
2	$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
3	$E \rightarrow T$	...
4	$T \rightarrow T_1 * F$	...
5	$T \rightarrow F$	...
6	$F \rightarrow (E)$	...
7	$F \rightarrow$ *digit*	F.val = *digit*.lexval

Parse Tree

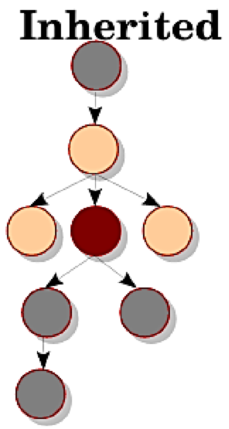


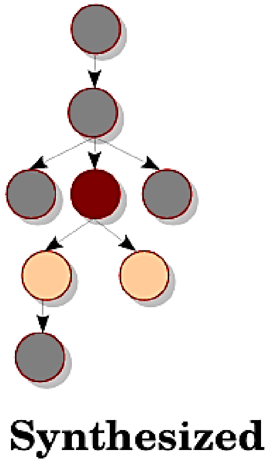
Annotated Parse Tree



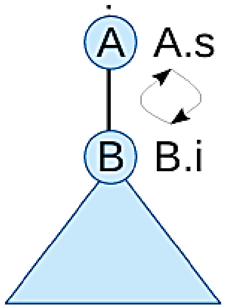
Order of Evaluation

- If there are only synthesized attributes in the SDD, there exists an evaluation order.
- Any bottom-up order would do; for instance, post-order.
- Helpful for LR parsing.
- How about when the attributes are both synthesized as well as inherited?
- How about when the attributes are only inherited?

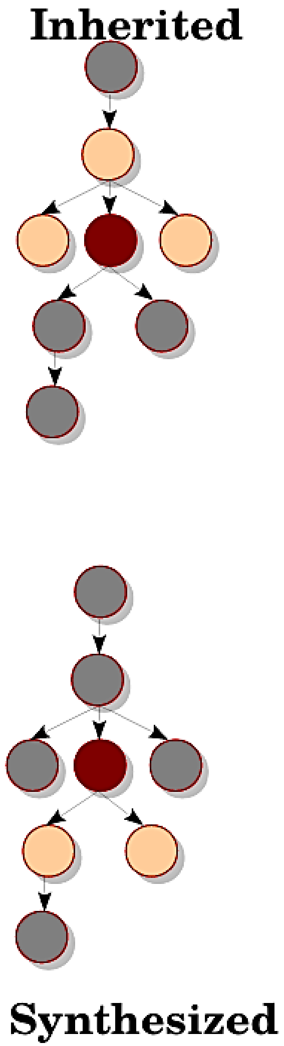




Order of Evaluation



Production	Semantic Rule
A → B	A.s = B.i; B.i = A.s + 1;



- This SDD uses a combination of synthesized and inherited attributes.
- A.s (head) is defined in terms of B.i (body non-terminal). Hence, it is synthesized.
- B.i (body non-terminal) is defined in terms of A.s (head). Hence, it is inherited.
- There exists a *circular dependency* between their evaluations.
- In practice, subclasses of SDDs required for our purpose do have an order.

Concrete Syntax Tree (CST)

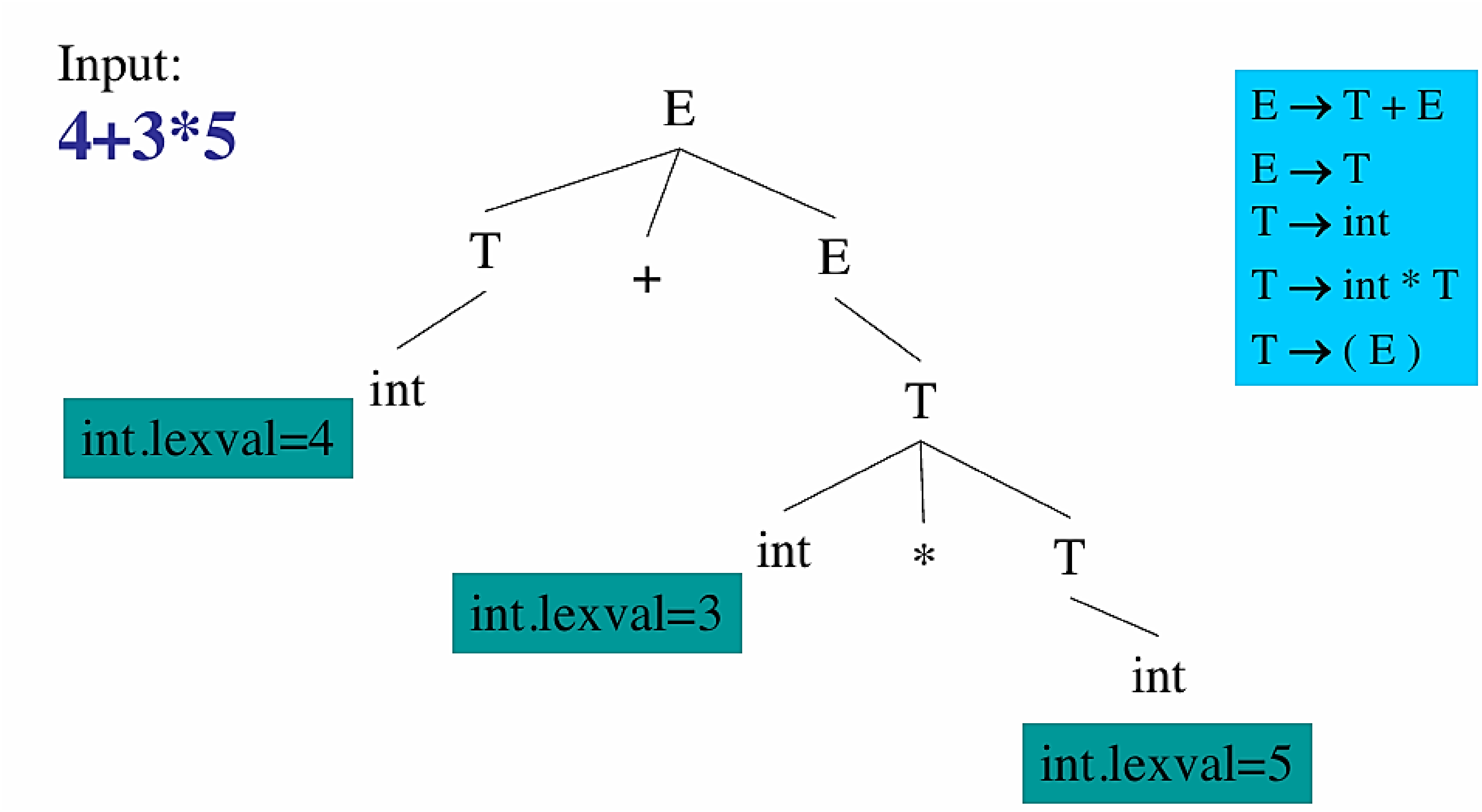
- In compiler design, a concrete syntax tree (CST), also known as a parse tree, is a tree-like representation of the syntactic structure of an input program, such as an expression, as defined by a formal grammar.
- It explicitly shows all the details of the grammar rules used to derive the input, including non-terminal symbols and the precise arrangement of terminal symbols (tokens).

```
Exp ::= Exp + Term | Term
Term ::= Term * Factor | Factor
Factor ::= (Exp) | id | num
```

For the expression `a + b * c`, the CST would explicitly show the derivation steps, including the `Exp`, `Term`, and `Factor` non-terminals, as well as the `+` and `*` operators and the `id` terminals.

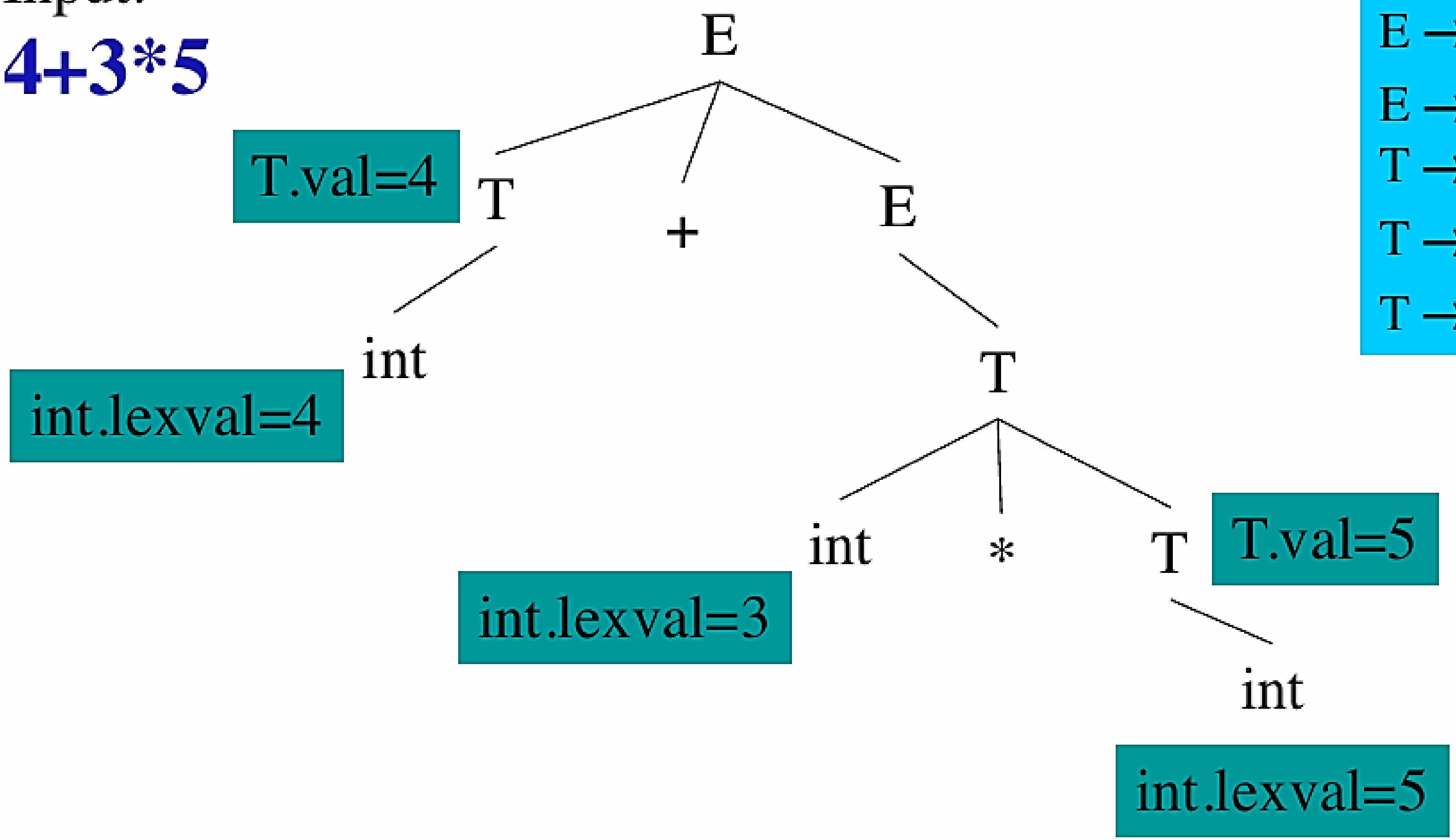
This level of detail distinguishes it from an Abstract Syntax Tree (AST), which would present a more reduced, semantically focused representation.

Expr Concrete Syntax Tree



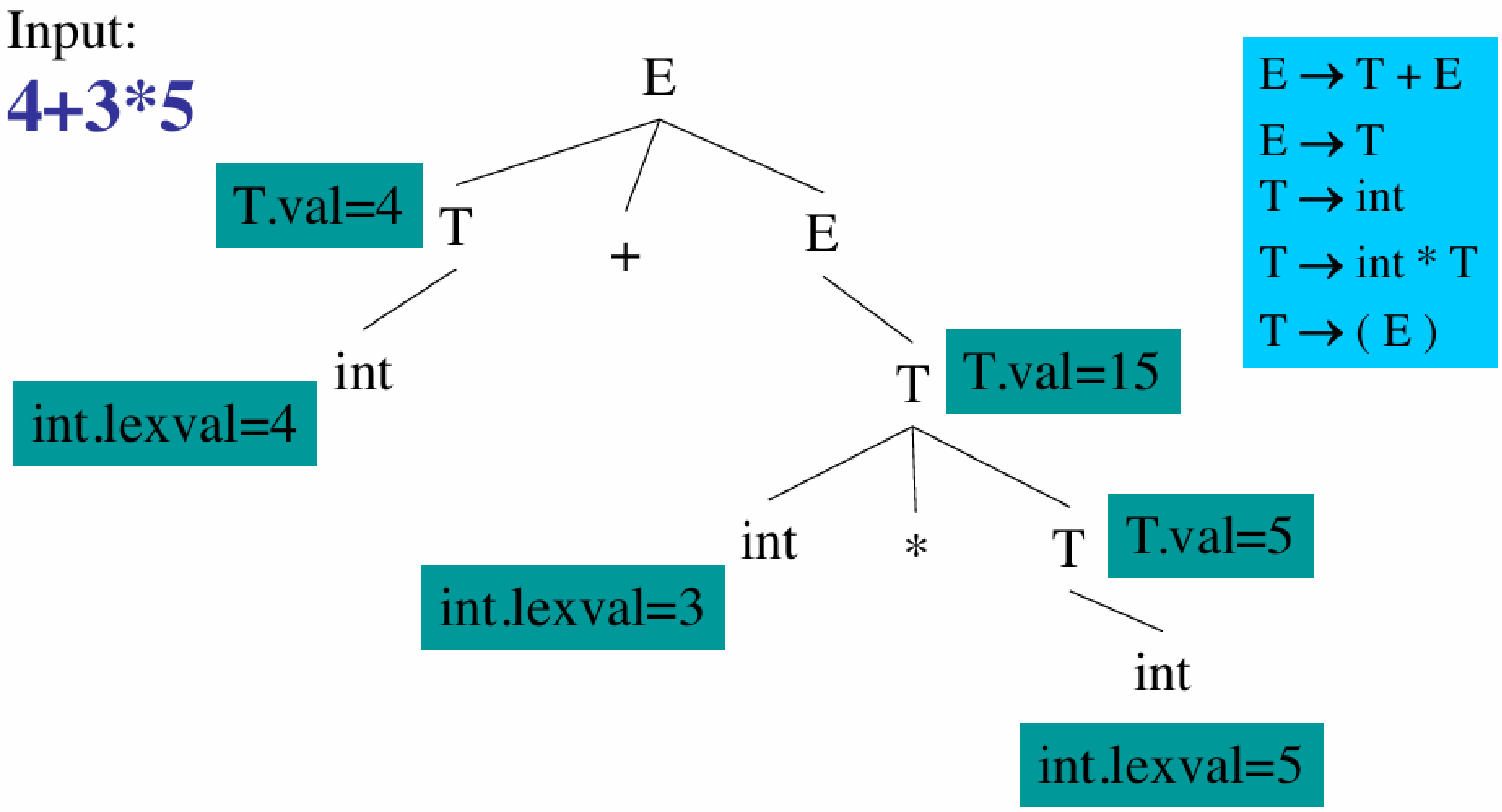
Expr Concrete Syntax Tree

Input:
4+3*5

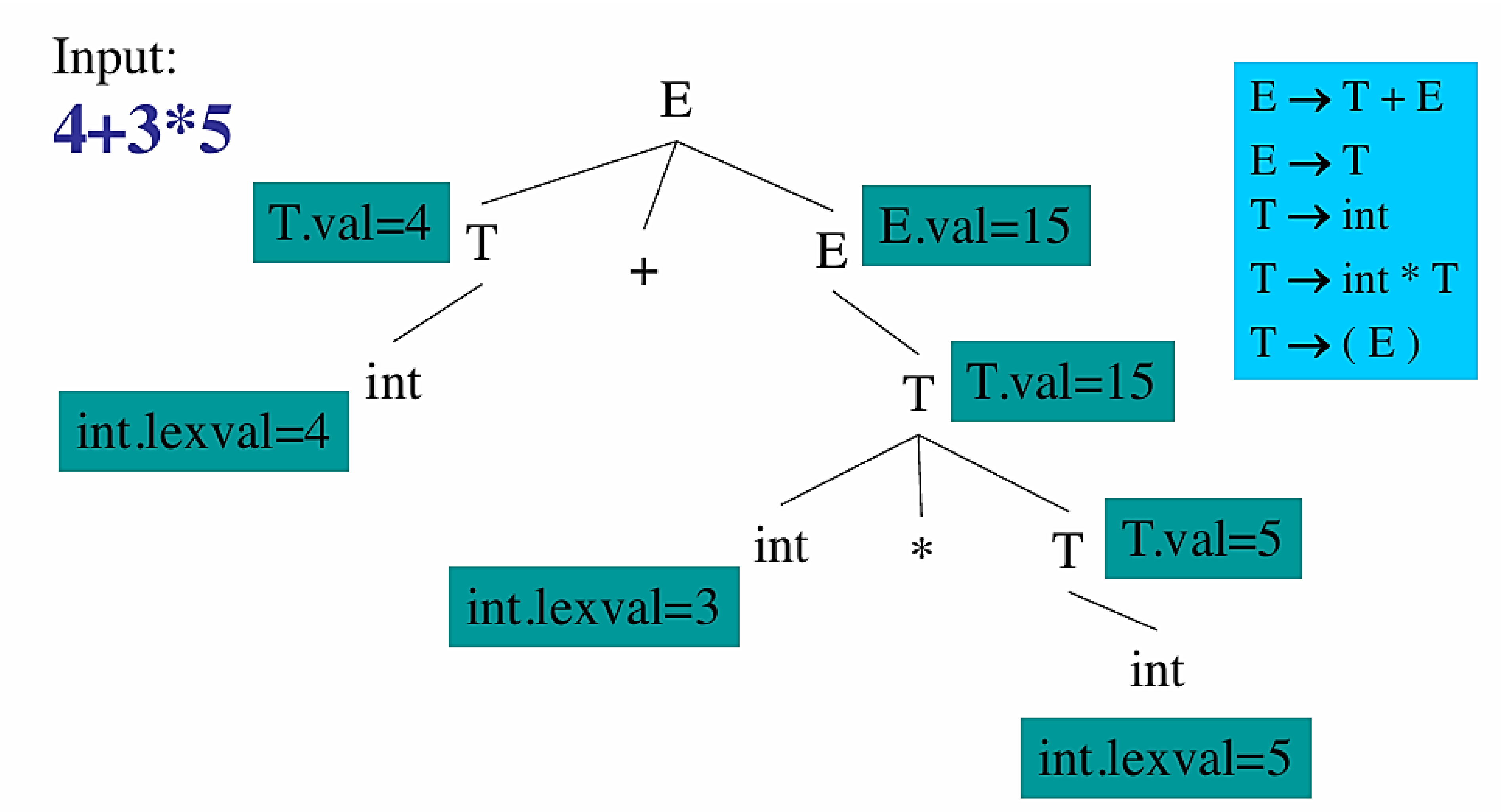


E → T + E
E → T
T → int
T → int * T
T → (E)

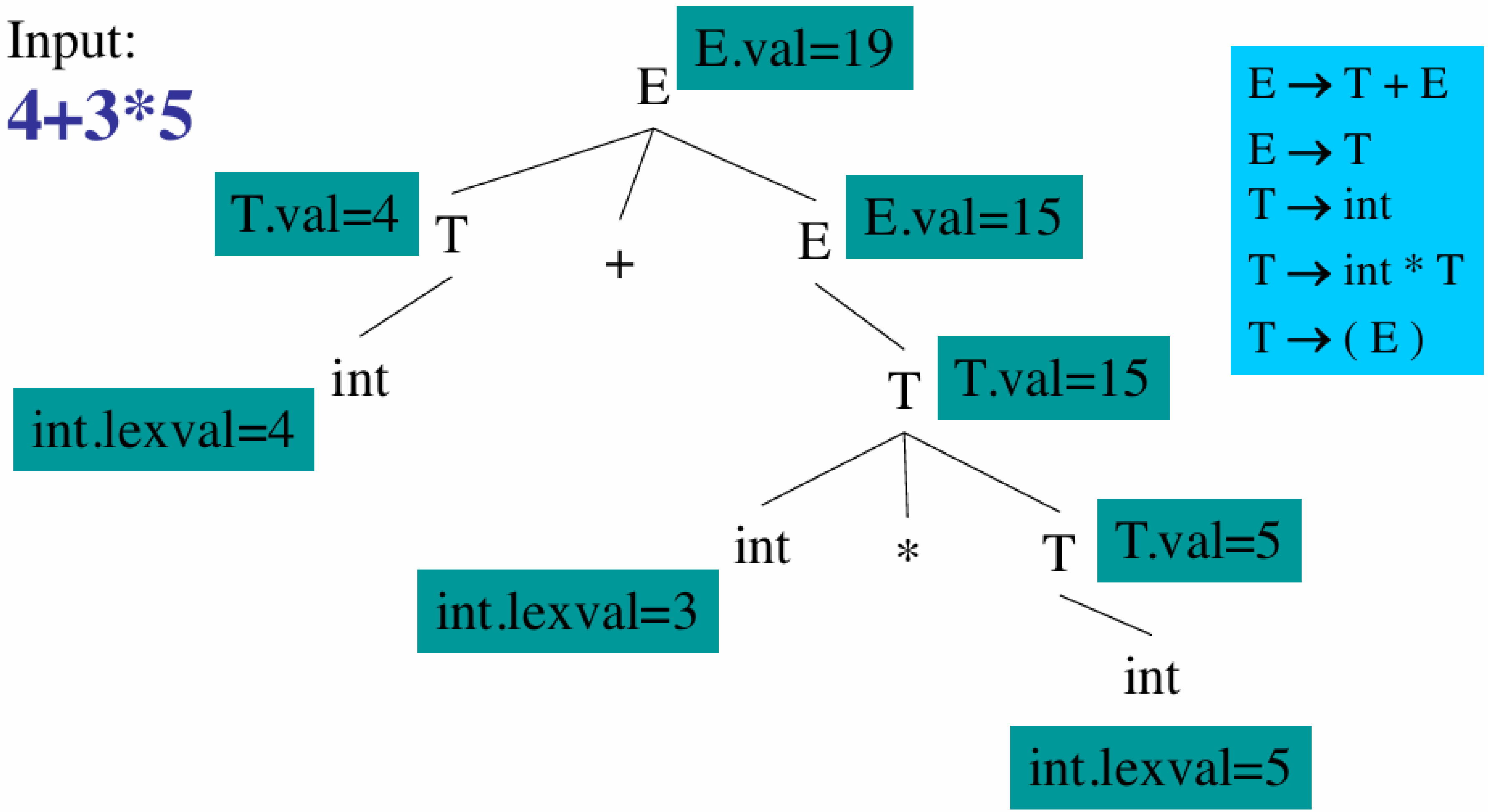
Expr Concrete Syntax Tree



Expr Concrete Syntax Tree



Expr Concrete Syntax Tree



- SDD = **Grammar + Attributes + Rules to compute them**

Syntax directed definition

The terminal `int` carries a lexical value (`lexval`) from the lexer (e.g., 4, 3, or 5). The nonterminal `T` simply takes that numeric value.

$T \rightarrow$ **int**

{ $0.val = $1.lexval; }

In yacc:
action is written as { $$ = $1 }
i.e. $0 == $$

$T \rightarrow$ **int** * T

{ $0.val = $1.lexval * $3.val ; }

This computes multiplication. The `first int` gives a number `($1.lexval)`, and `$3.val` gives the result of the next `T`.

$E \rightarrow T$

{ $0.val = $1.val; }

An expression `(E)` that's just a term `(T)` has the same value as that term.

$E \rightarrow T$ **+** E

{ $0.val = $1.val + $3.val; }

This rule performs addition: left term `($1.val)` + right expression `($3.val)`.

$T \rightarrow$ **(** E **)**

{ $0.val = $2.val; }

Parentheses don't change value — just pass the value of the inner expression.

Attribute Evaluation (Bottom-Up)

Node	Production Used	Computation	Result
int.lexval=4	T → int	$0.val = 4	T.val=4
int.lexval=3 and int.lexval=5	T → int * T	$0.val = 3 * 5	T.val=15
T.val=15	E → T	$0.val = 15	E.val=15
Top-level	E → T + E	$0.val = 4 + 15	E.val=19

Final result: `E.val = 19`

Summary of SDD

Concept	Meaning
Attributes	Store information like values or types for grammar symbols.
Synthesized attribute	Computed from child nodes (e.g., E.val = T.val + E.val).
Lexval	Lexical value provided by the lexer for tokens like integers.
SDD Rule Format	{ $$ = operation($1, $2, $3, ...) } — defines how parent attributes are computed.
Evaluation Order	Bottom-up in parse tree (post-order traversal).

Flow of Attributes in *Expr*

- Consider the flow of the attributes in the E syntax-directed defn
  - The lhs attribute is computed using the rhs attributes
- Purely bottom-up:
  - compute attribute values of all children (rhs) in the parse tree
  - And then use them to compute the attribute value of the parent (lhs)

Synthesized Attributes

- Synthesized attributes are attributes that are computed purely bottom-up
- A grammar with semantic actions (or syntax-directed definition) can choose to use only synthesized attributes
- Such a grammar plus semantic actions is called an S-attributed definition
- Synthesized attributes may not be sufficient for all cases that might arise for semantic checking and code generation.
- *Consider the (sub)grammar:*

Var-decl $\rightarrow$ Type Id-comma-list **;**

Type $\rightarrow$ **int** | **bool**

Id-comma-list $\rightarrow$ **ID**

Id-comma-list $\rightarrow$ **ID ,** Id-comma-list

Syntax-Directed Definition (SDD) example, but this time for **variable declarations**

Contd.,

Var-decl → Type Id-comma-list **;**

Type → **int** | **bool**

Id-comma-list → **ID**

Id-comma-list → **ID ,** Id-comma-list

Goal: **propagate the type** (int or bool) declared in the Type nonterminal **to all identifiers** (ID) in the list.

That means every variable on the left-hand side of the declaration inherits the declared type.

Attribute	Type	Meaning
Type.val	Synthesized	Stores the type value returned by Type (either int or bool).
Id-Comma-List.in	Inherited	Passes the declared type from the Type node down to each identifier in the list.
ID.val	Synthesized	Assigned from the inherited type (in) value.

Contd.,

Production	Semantic Rule
Var-decl → Type Id-Comma-List ;	Id-Comma-List.in = Type.val
Type → int	Type.val = int
Type → bool	Type.val = bool