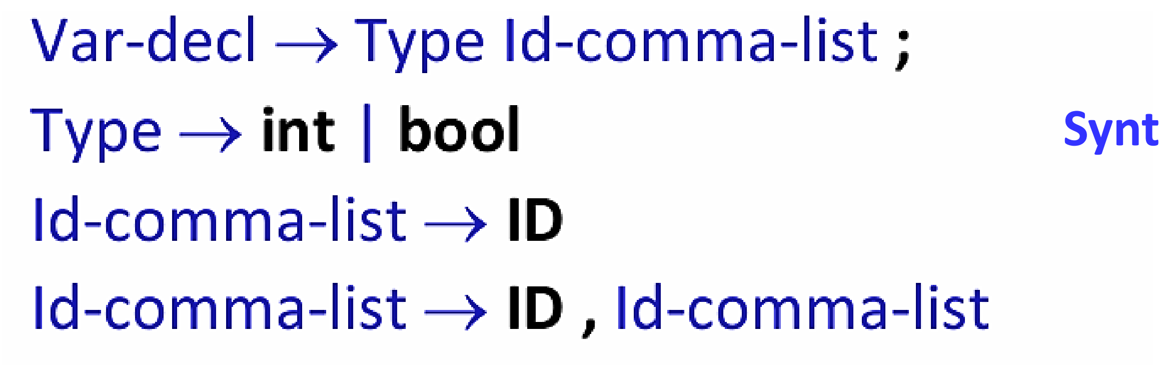
Synthesized Attributes

- 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 grammar:*



Var-decl → Type Id-comma-list **;**

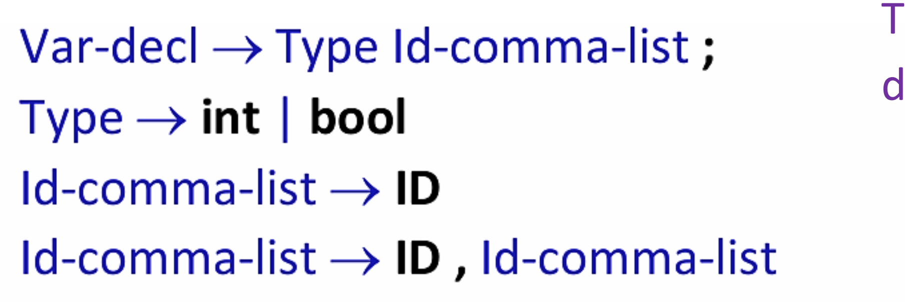
Type → **int** | **bool**

Id-comma-list → **ID**

Id-comma-list → **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

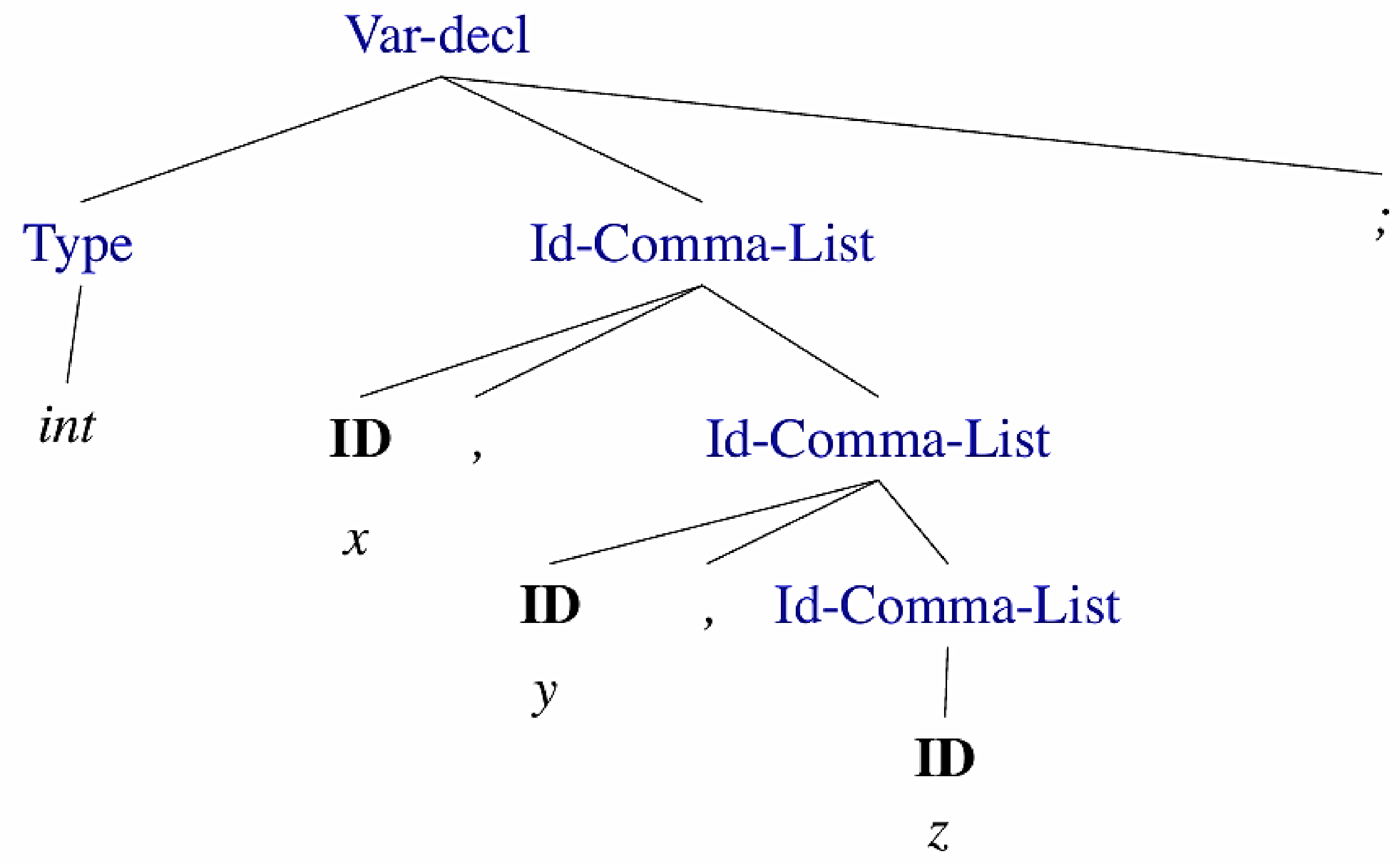
Goal: **broadcast the type** (int or bool) declared in the Type nonterminal **to all identifiers** (ID) in the list.

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.

Var-decl → Type Id-comma-list **;**
Type → **int** | **bool**
Id-comma-list → **ID**
Id-comma-list → **ID ,** Id-comma-list

Example input: int x, y, z ;



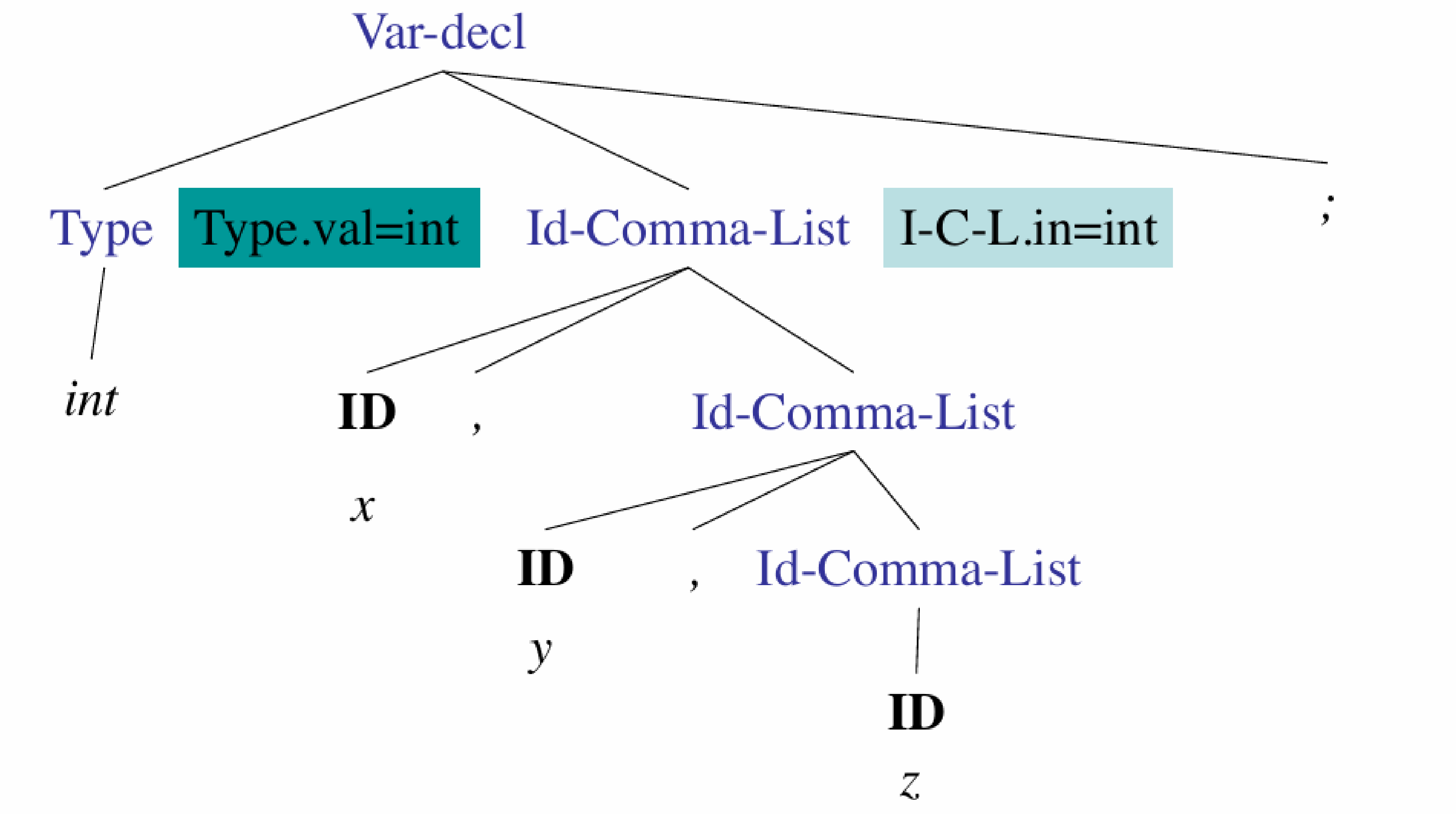
Example input: int x, y, z ;

Var-decl → Type Id-comma-list **;**

Type → **int** | **bool**

Id-comma-list → **ID**

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



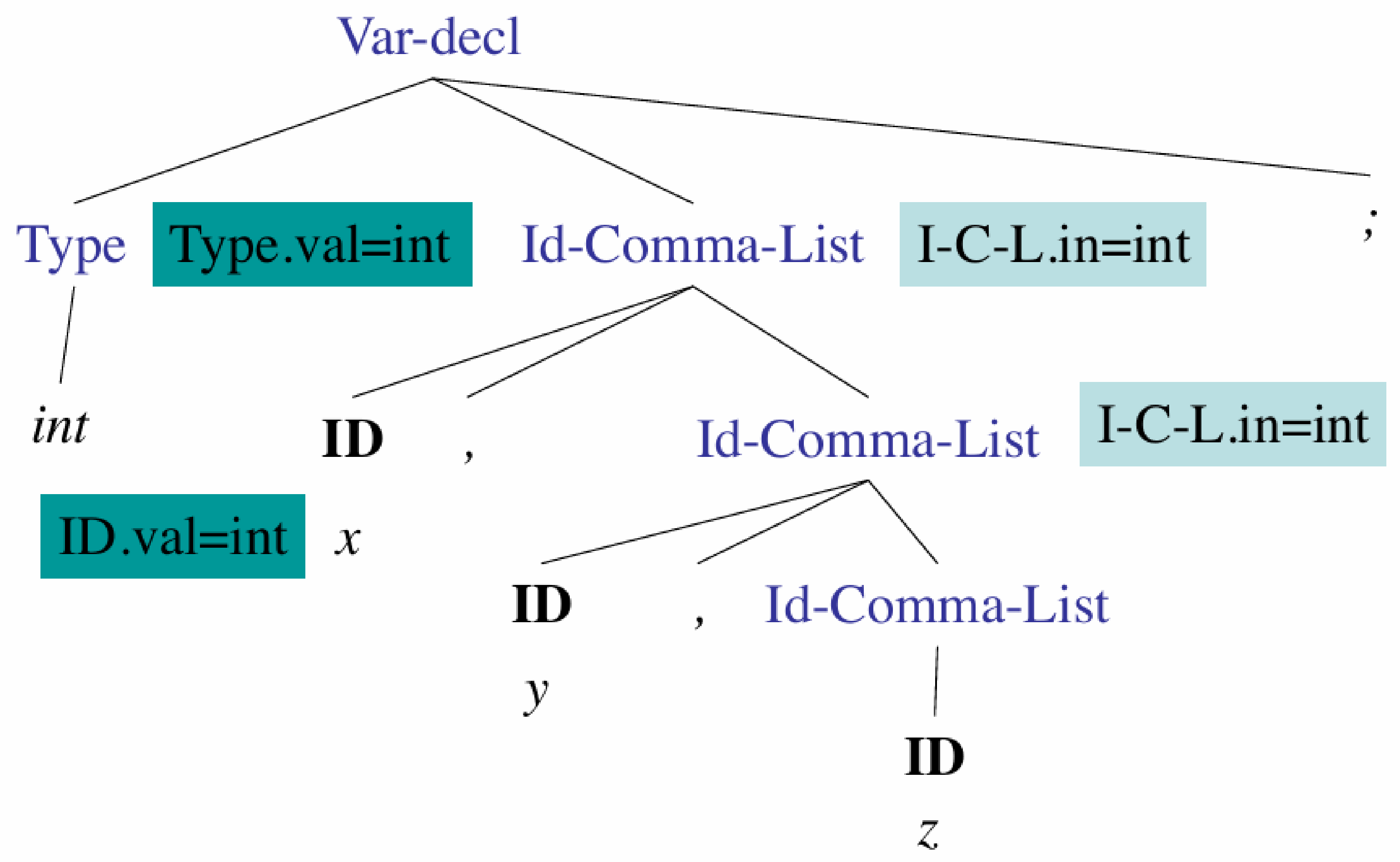
Example input: int x, y, z ;

Var-decl → Type Id-comma-list **;**

Type → **int** | **bool**

Id-comma-list → **ID**

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



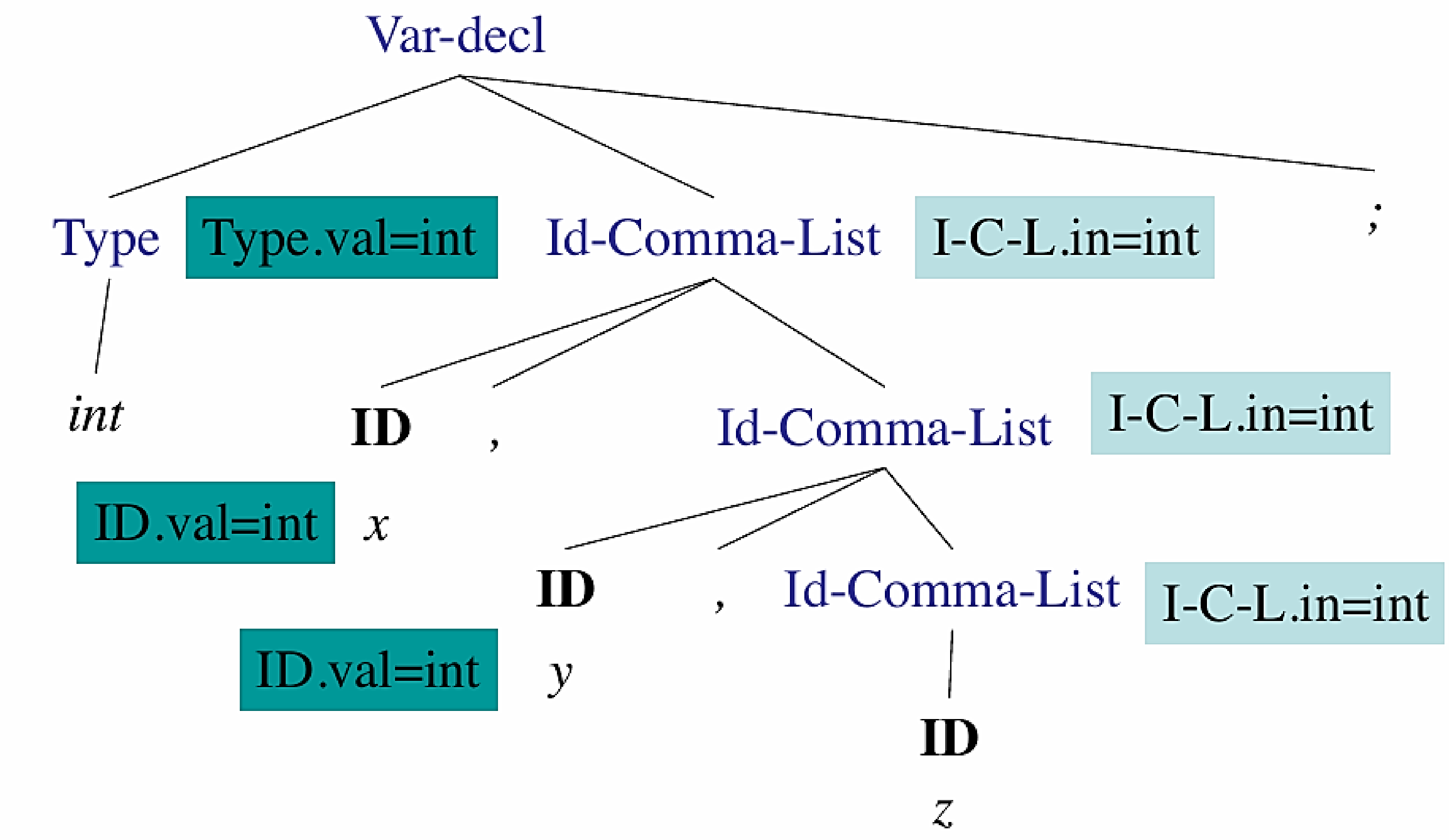
Example input: int x, y, z ;

Var-decl → Type Id-comma-list **;**

Type → **int** | **bool**

Id-comma-list → **ID**

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



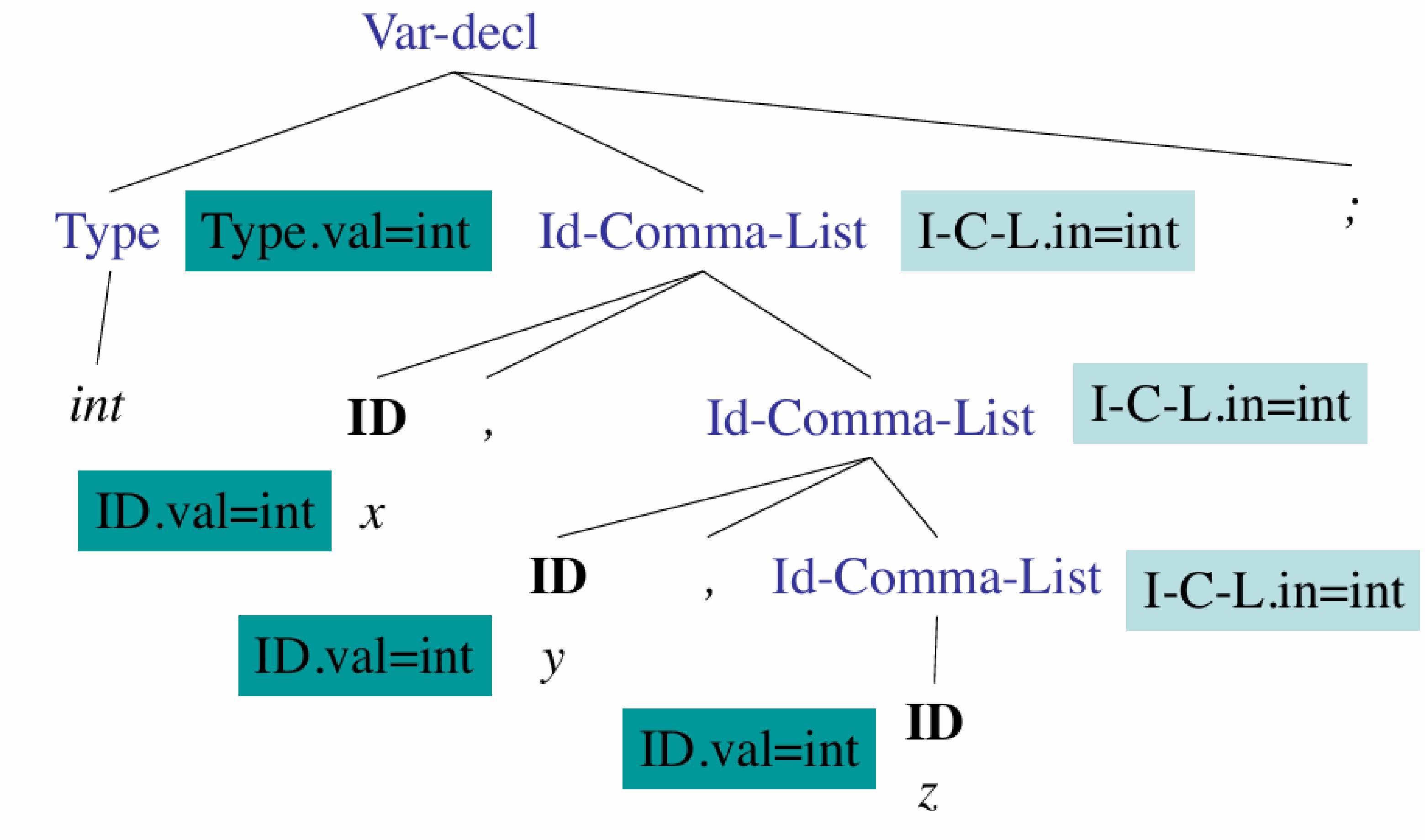
Example input: int x, y, z ;

Var-decl → Type Id-comma-list **;**

Type → **int** | **bool**

Id-comma-list → **ID**

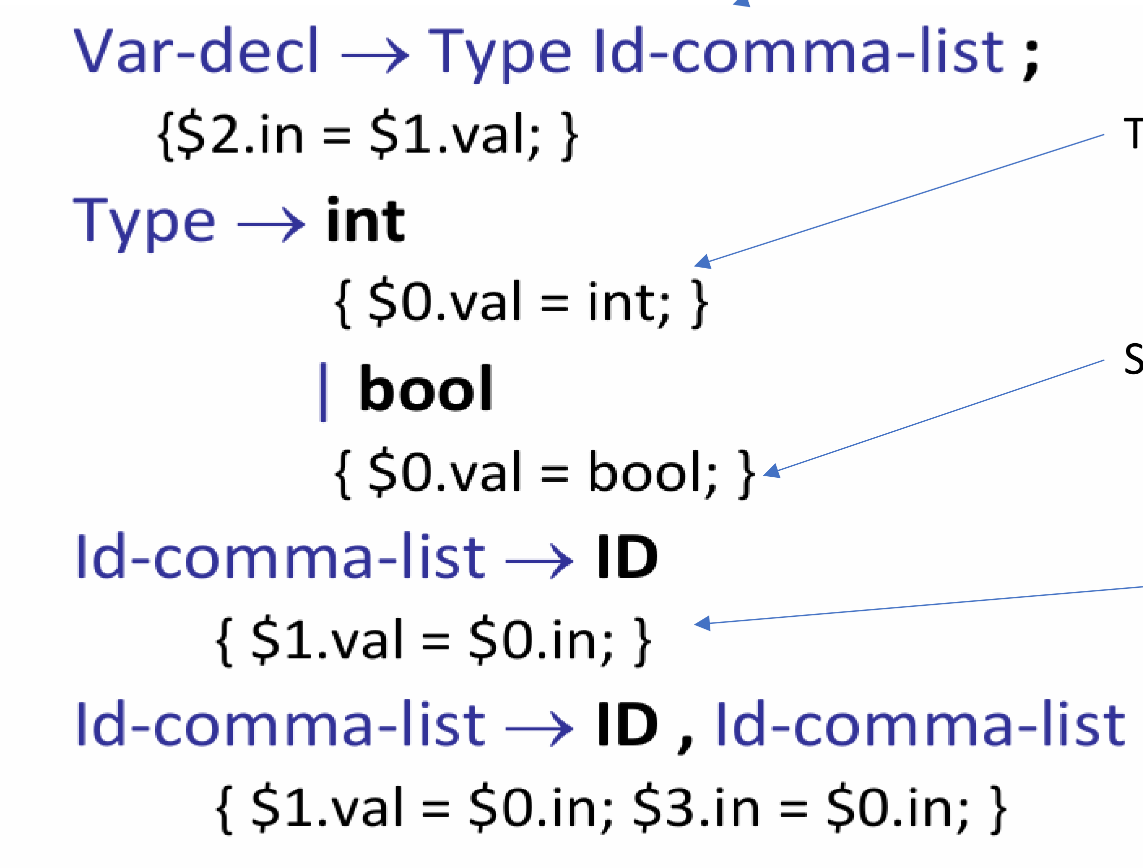
Id-comma-list → **ID ,** Id-comma-list



Flow of Attributes in Var-decl

- How do the attributes flow in the Var - decl grammar?
- ID takes its attribute value from its parent node
- IdList takes its attribute from its left sibling Type
- or IdList takes its attribute from its parent IdList

Syntax-directed definition



```
Var-decl → Type Id-comma-list ;
    {$2.in = $1.val; }
Type → int
            { $0.val = int; }
          | bool
            { $0.val = bool; }
Id-comma-list → ID
      { $1.val = $0.in; }
Id-comma-list → ID , Id-comma-list
      { $1.val = $0.in; $3.in = $0.in; }
```

The type (**either `int` or `bool`**) obtained from **`Type`** is **passed down (inherited)** to `Id-comma-list`. So if `Type.val = int`, then `$2.in = int`.

The keyword `int` gives the synthesized attribute `val = int`.

Similarly, for `bool`, the synthesized attribute `val = bool`.

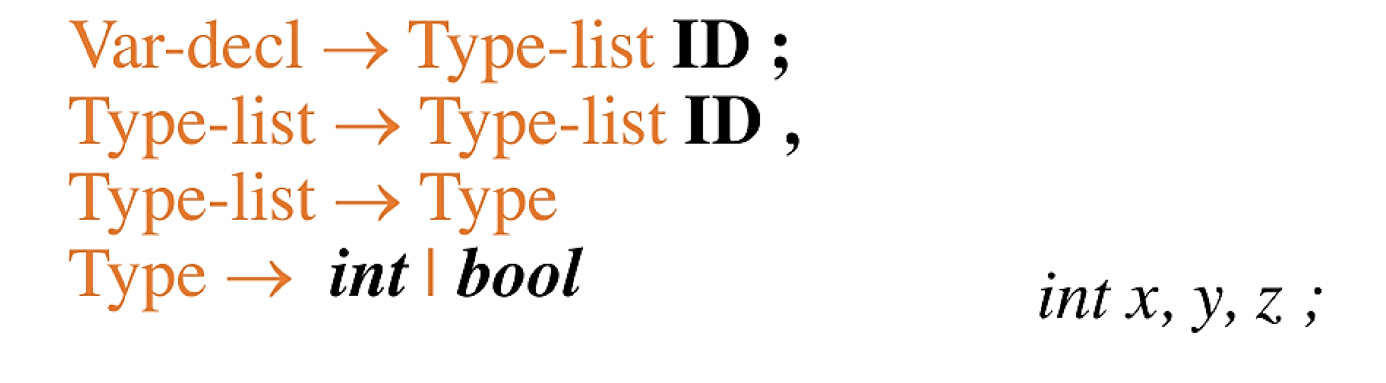
The inherited attribute `in` from the parent list is assigned to the identifier. Example: if **`in = int`**, then **`$1.val = int`**.

Both the first ID and the next list of IDs inherit the same declared type. Example: for `int x, y, z;` all IDs get `int`.

{ $2.in = $1.val; } means “set the second RHS symbol’s inherited attribute equal to the first RHS symbol’s synthesized attribute.”

Inherited Attributes

- Inherited attributes are computed at a node based on attributes from siblings or the parent
- Typically, we combine synthesized attributes and inherited attributes
- **Q**: It is possible to convert the grammar into a form that only uses synthesized attributes?



Var-decl → Type-list **ID ;**
Type-list → Type-list **ID ,**
Type-list → Type
Type → ***int*** | ***bool***

int x, y, z ;

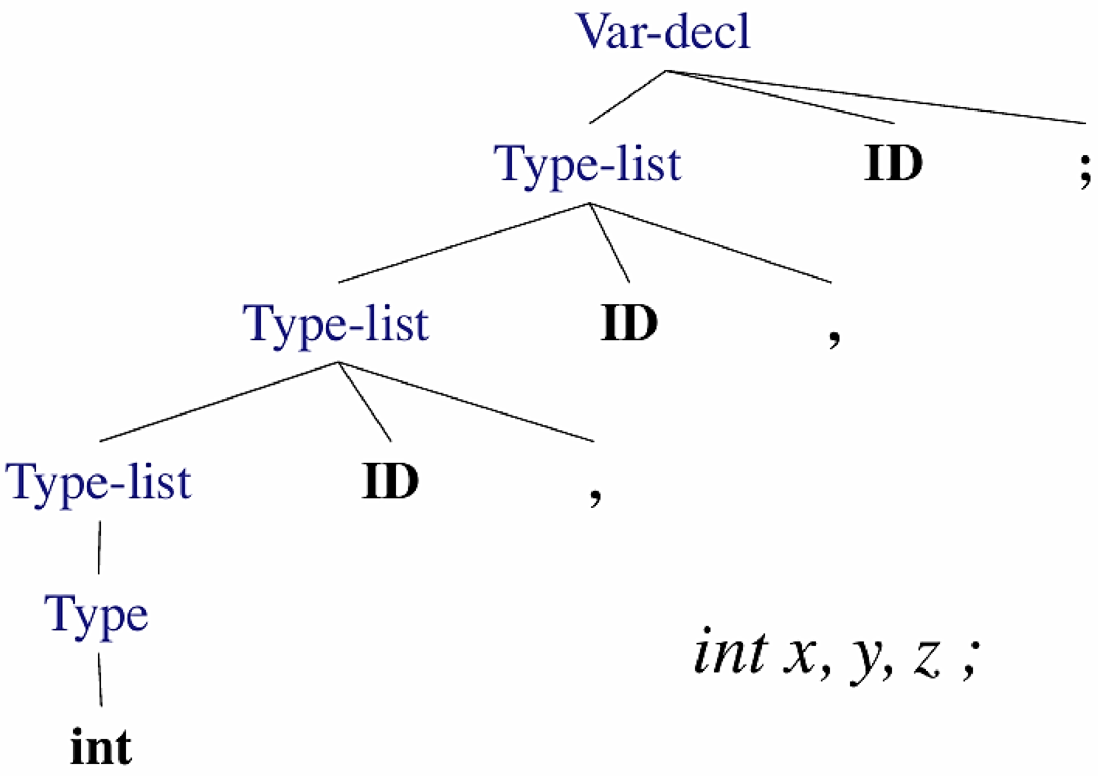
"Remove Inherited Attributes"? But Why

- In **Syntax-Directed Definitions (SDDs)**:
  - **Inherited attributes** pass information **downward** (parent → child).
    Example: The declared type (int or bool) is passed to each variable name.
  - **Synthesized attributes** pass information **upward** (child → parent).
- **But** *inherited attributes make bottom-up parsing (like LR parsers) harder*, because:
- Bottom-up parsers build parse trees **from leaves upward**, so inherited information isn't known yet when a child is processed.
- Therefore, we eliminate inherited attributes by **restructuring the grammar** so that all needed information is synthesized.

Removing Inherited Attributes

Var-decl → Type-list **ID ;**
Type-list → Type-list **ID ,**
Type-list → Type
Type → ***int*** | ***bool***

int x, y, z ;

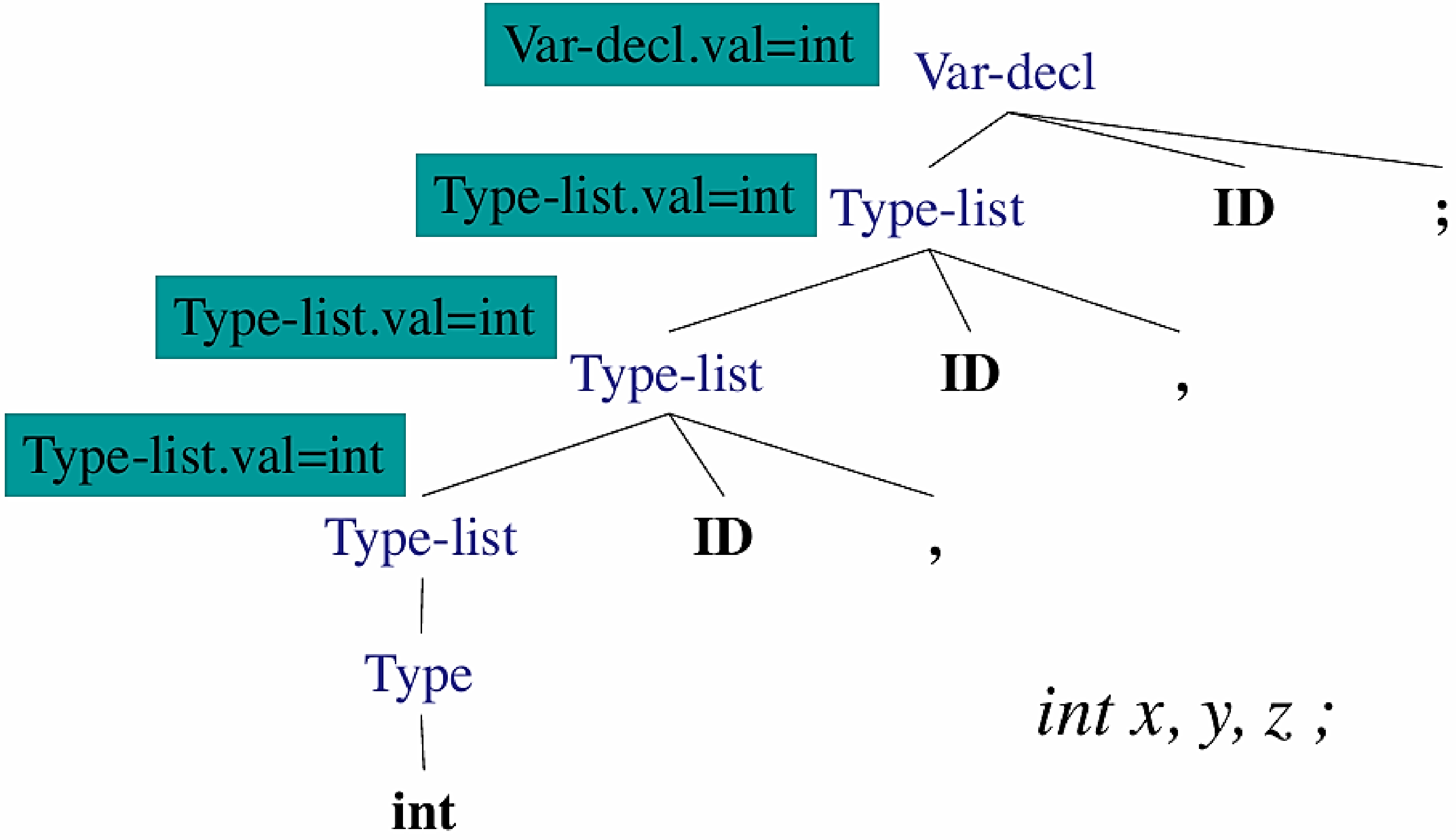


int x, y, z ;

Removing Inherited Attributes

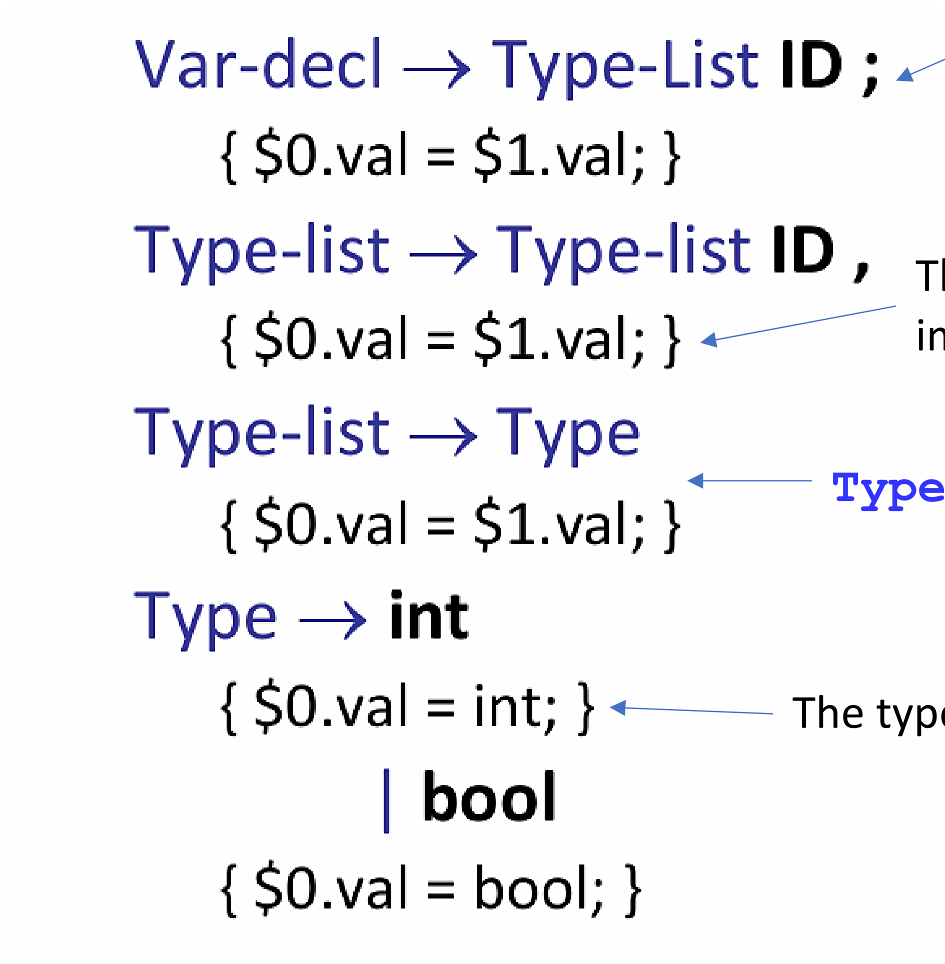
Var-decl → Type-list **ID ;**
Type-list → Type-list **ID ,**
Type-list → Type
Type → ***int*** | ***bool***

int x, y, z ;



int x, y, z ;

Removing inherited attributes



```
Var-decl → Type-List ID ;
    { $0.val = $1.val; }
Type-list → Type-list ID ,
    { $0.val = $1.val; }
Type-list → Type
    { $0.val = $1.val; }
Type → int
    { $0.val = int; }
        | bool
    { $0.val = bool; }
```

The entire variable declaration now has: **int** and thus, **all variables x, y, z are understood to be of type int.**

The left `Type-list` keeps the same **`val = int`**, and the ID (e.g., `x`) uses it implicitly (repeats for y and z).

`Type-list` now carries the type **`"int"`** upward as a **synthesized** attribute.

The type value **`"int"`** is synthesized and stored as **`Type.val`**.

Read from bottom → synthesized attribute

Contd., Summary

Production	Semantic Rule	Explanation
Var-decl → Type-list ID ;	{ $0.val = $1.val; }	Propagate type upward.
Type-list → Type-list ID ,	{ $0.val = $1.val; }	Keep the same type for next ID.
Type-list → Type	{ $0.val = $1.val; }	Initialize type from the keyword.
Type → int	{ $0.val = int; }	Define the type.
Type → bool	{ $0.val = bool; }	Define another type.

"Instead of passing the declared type *downward*, we make it flow *upward* through each nonterminal, ensuring the same information is preserved."

Direction of inherited attributes

- Consider the syntax directed definations
  - **A → L M**
  - { \$1.in = \$0.in; \$2.in = \$1.val; \$0.val = \$2.val; }

- **Explanation:**
  - \$1.in = \$0.in; → L (the first child) inherits the input from A.
  - \$2.in = \$1.val; → M inherits a value **from its left sibling (L)**.
  - \$0.val = \$2.val; → The result of A is taken from M.

- This is a **left-to-right** information flow.

- Meaning: data moves from **A → L → M** (which works well in most parsing methods).

Contd.,

  - A → Q R
  - { \$2.in = \$0.in; \$1.in = \$2.val; \$0.val = \$1.val; }

- **Explanation:**
    - \$2.in = \$0.in; → R (second child) inherits input from A.
    - \$1.in = \$2.val; → Q tries to get information **from its right sibling (R)**.
    - \$0.val = \$1.val; → A's result is taken from Q.

- **Problem:** \$1.in = \$2.val

- This means the left symbol (Q) depends on the **value of its right sibling (R)** — but in **top-down or left-to-right parsing**, R hasn't been processed yet.

Incremental Processing

- Incremental processing: constructing output as we are parsing

- Bottom-up or top-down parsing
  - Both can be viewed as left-to-right and depth-first construction of the parse tree

L-attributed Definitions

- A syntax-directed definition is L-attributed if for each production $A \rightarrow X_1..X_{j\ -1}\ X_j..X_n$, for each j=1 …n, each inherited attribute of Xj depends on:
  - – The attributes of $X_1...X_{j-1}$
  - – The inherited attributes of A
- These two conditions ensure left to right and depth first parse tree construction
- Every S-attributed definition is L-attributed

Syntax-directed defns

- Different SDTs are defined based on the parser that is used.

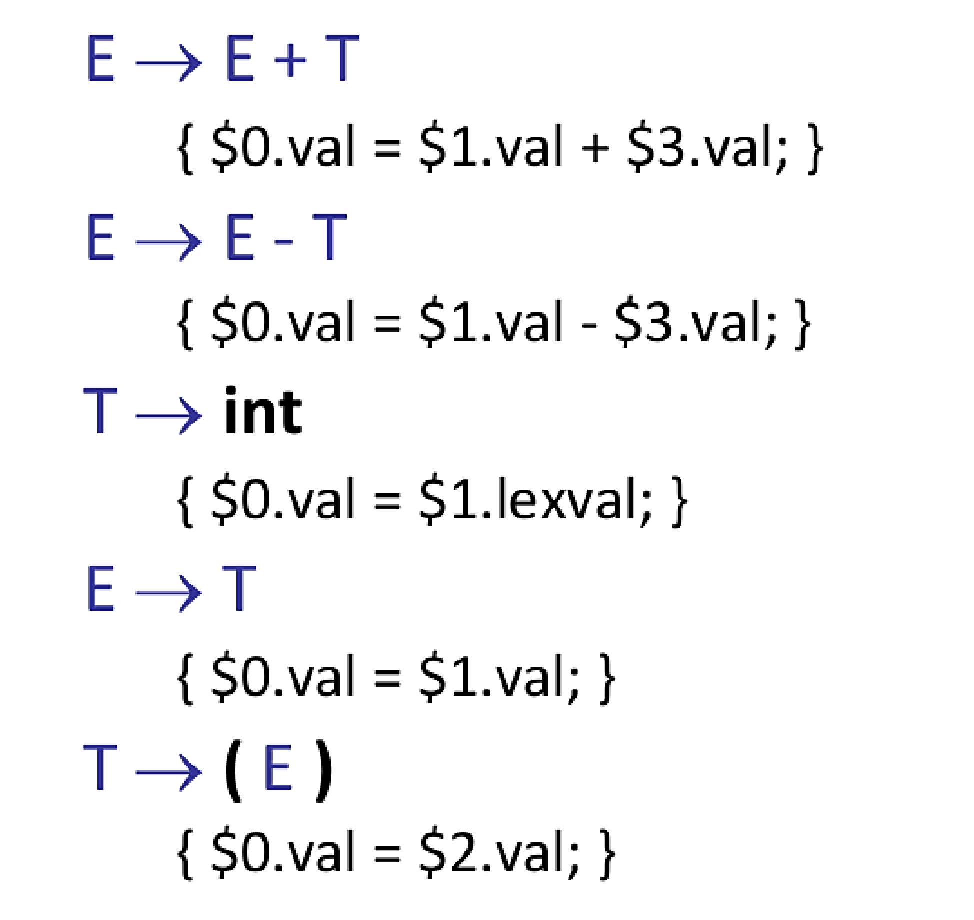
- LR parser, S-attributed definition
  - Implementing S-attributed definitions in LR parsing is easy: execute action on reduce, all necessary attributes have to be on the stack

- LL parser, L-attributed definition
  - Implementing L-attributed definitions in LL parsing: we need an additional action record for storing synthesized and inherited attributes on the parse stack

Top-down translation

- Assume that we have a top-down predictive parser
- Typical strategy: take the CFG and eliminate left-recursion
- Suppose that we start with an attribute grammar
- We should still eliminate left-recursion

Top-down translation example



E → E + T

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

E → E - T

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

T → **int**

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

E → T

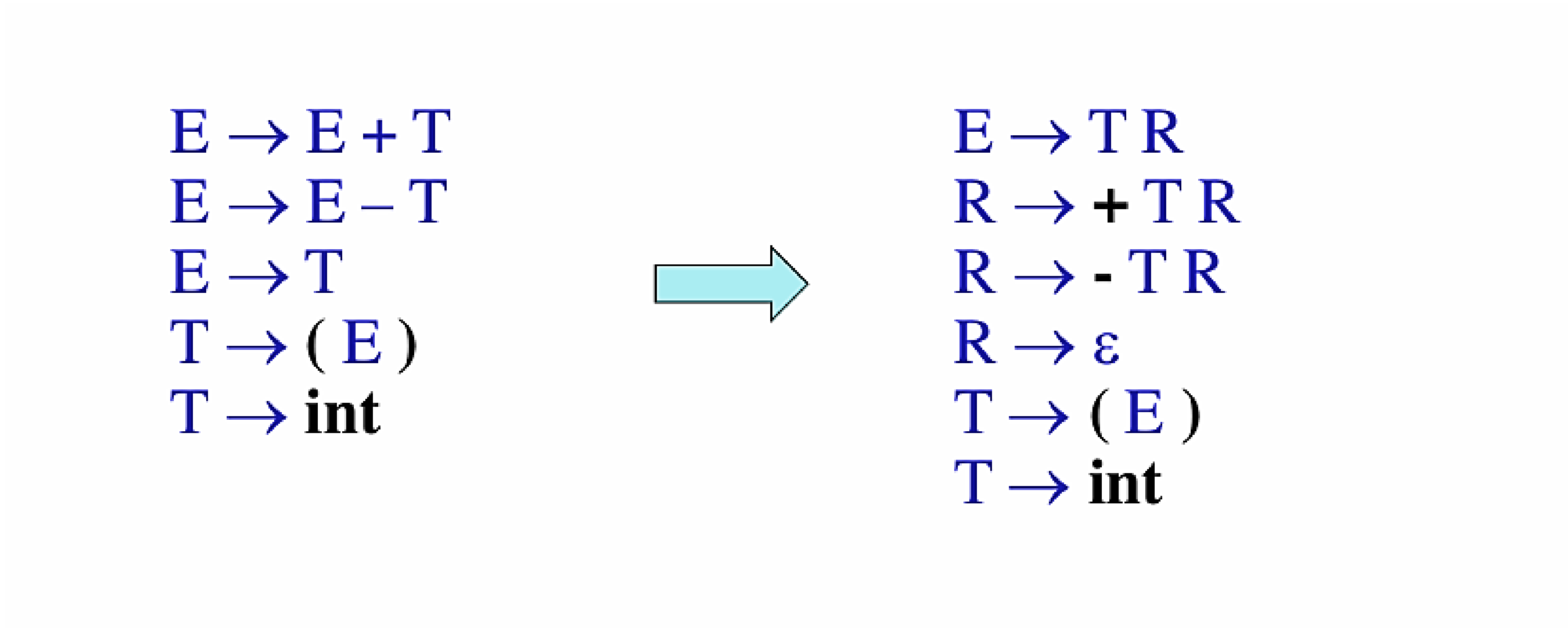
{ $0.val = $1.val; }

T → **(** E **)**

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

Top-down translation example

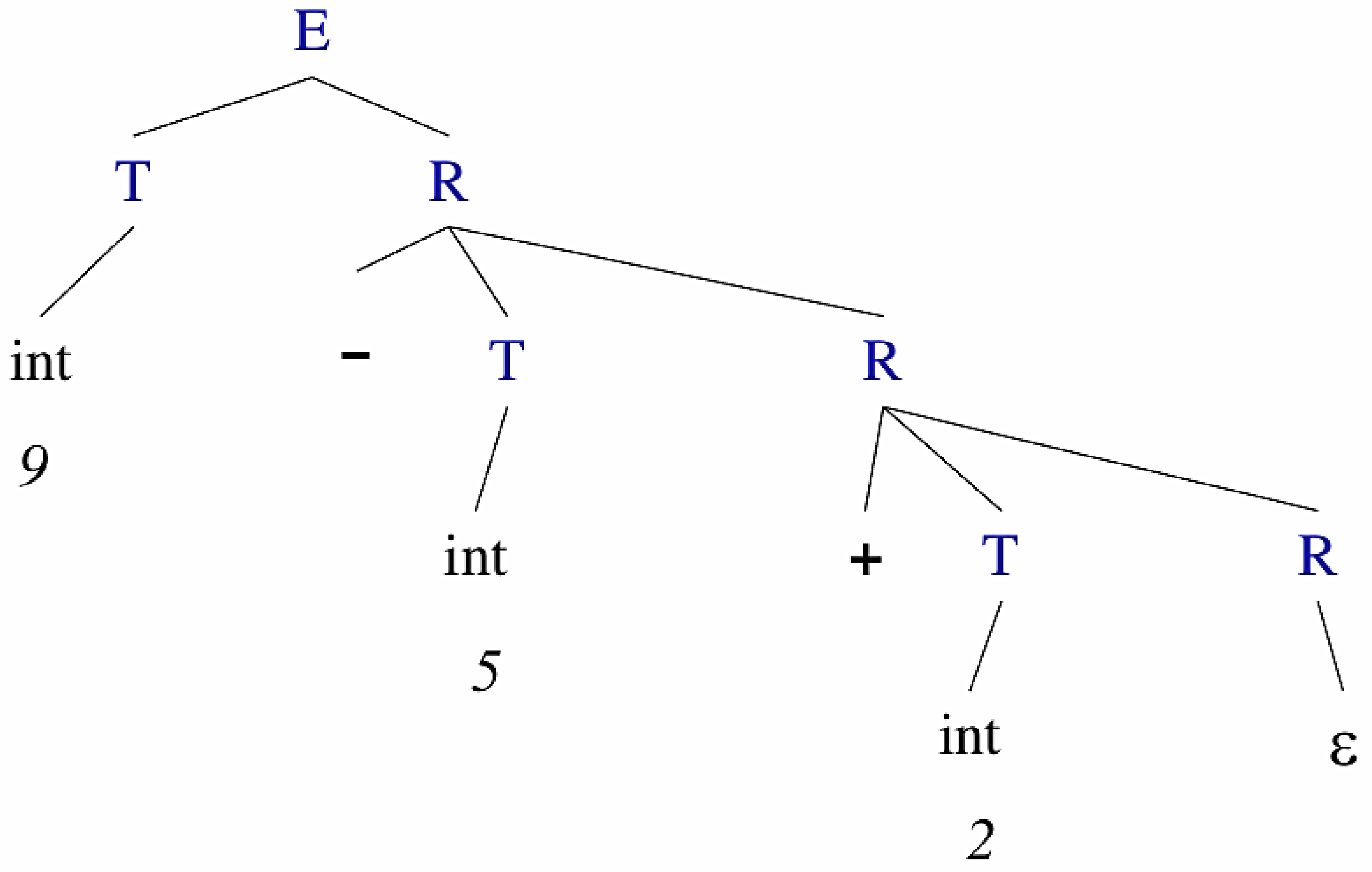
- Remove Left recursion



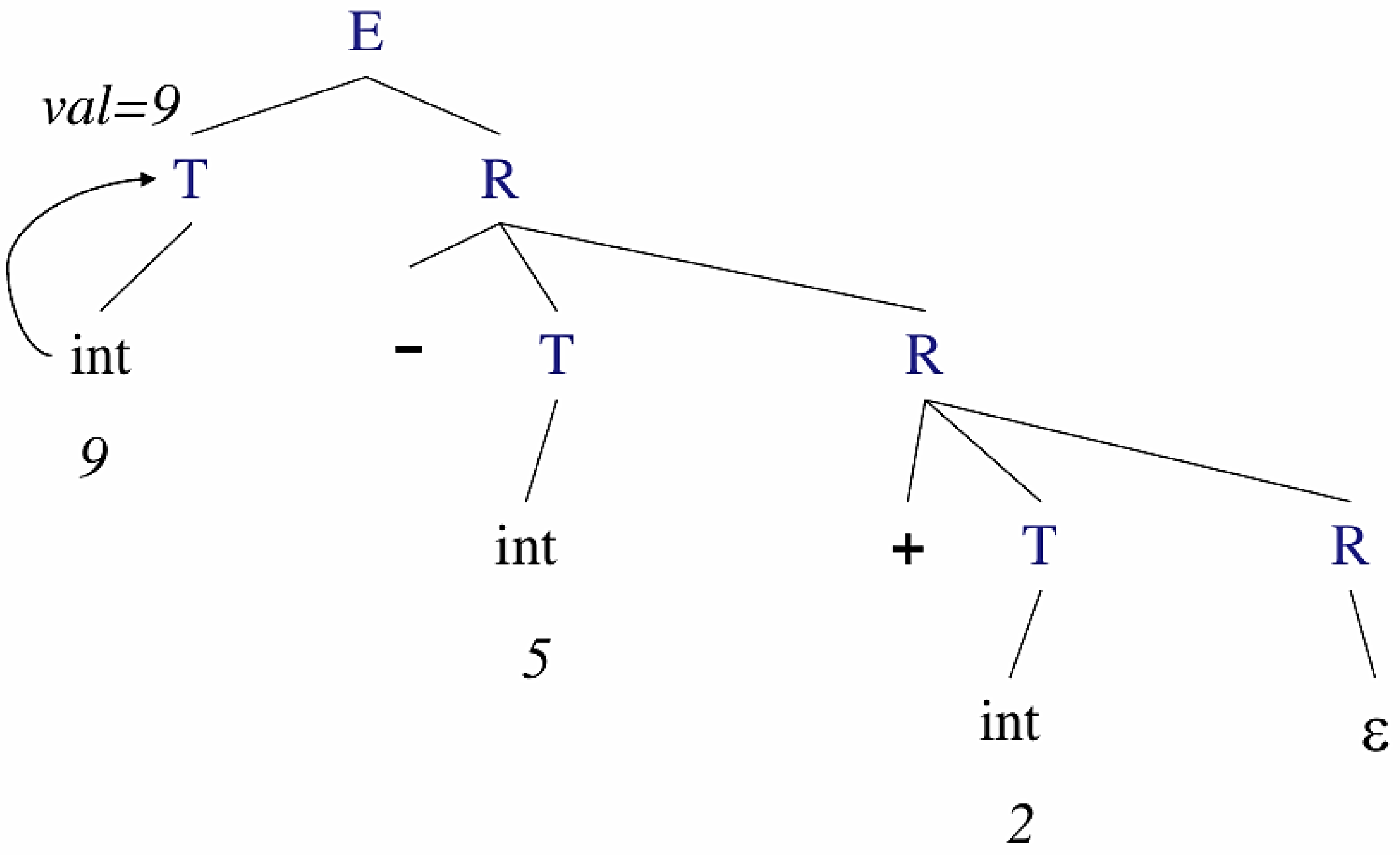
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow$ **int**

$E \rightarrow T\ R$
$R \rightarrow$ **+** $T\ R$
$R \rightarrow$ **-** $T\ R$
$R \rightarrow \varepsilon$
$T \rightarrow (E)$
$T \rightarrow$ **int**

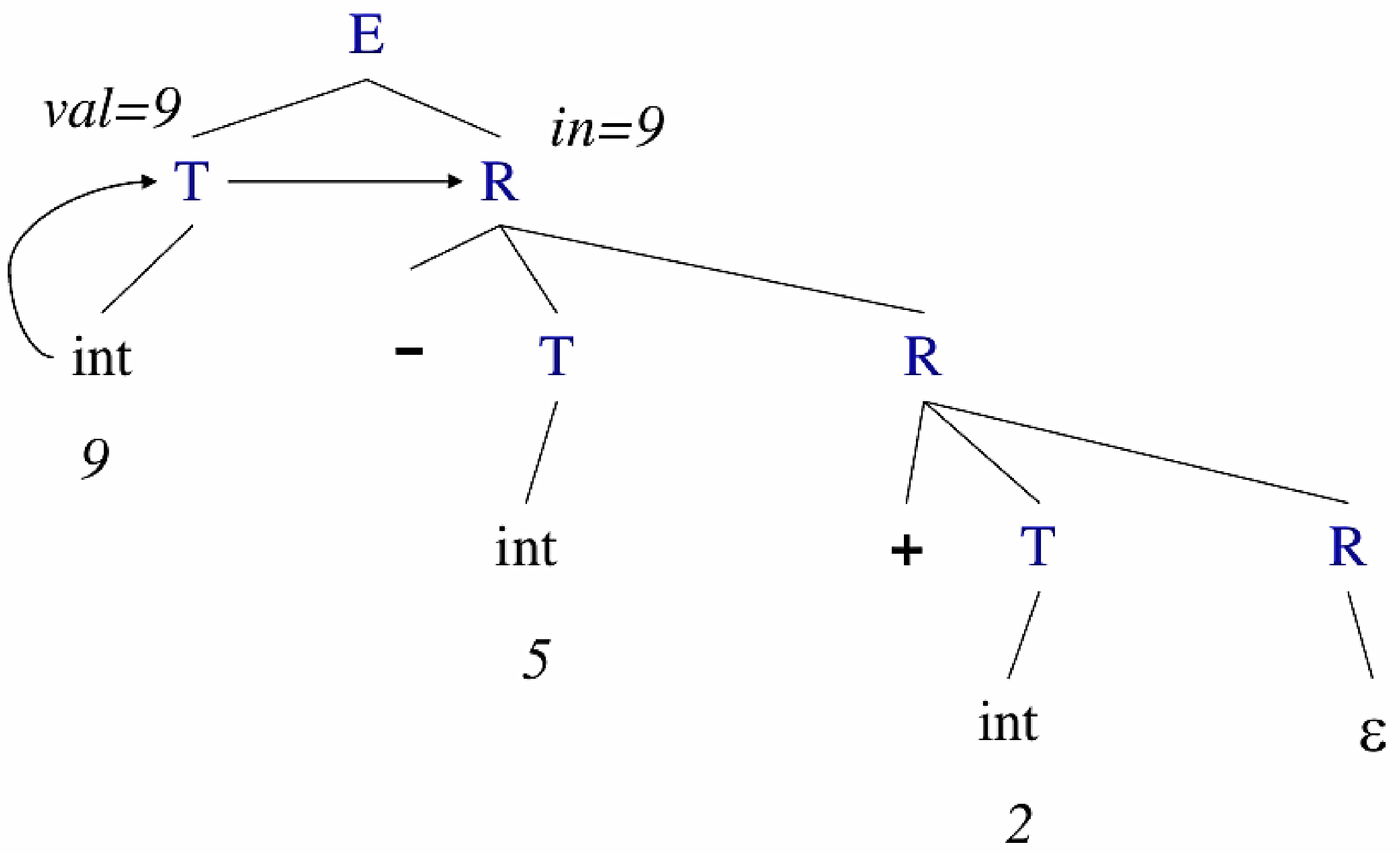
input: 9 - 5 + 2



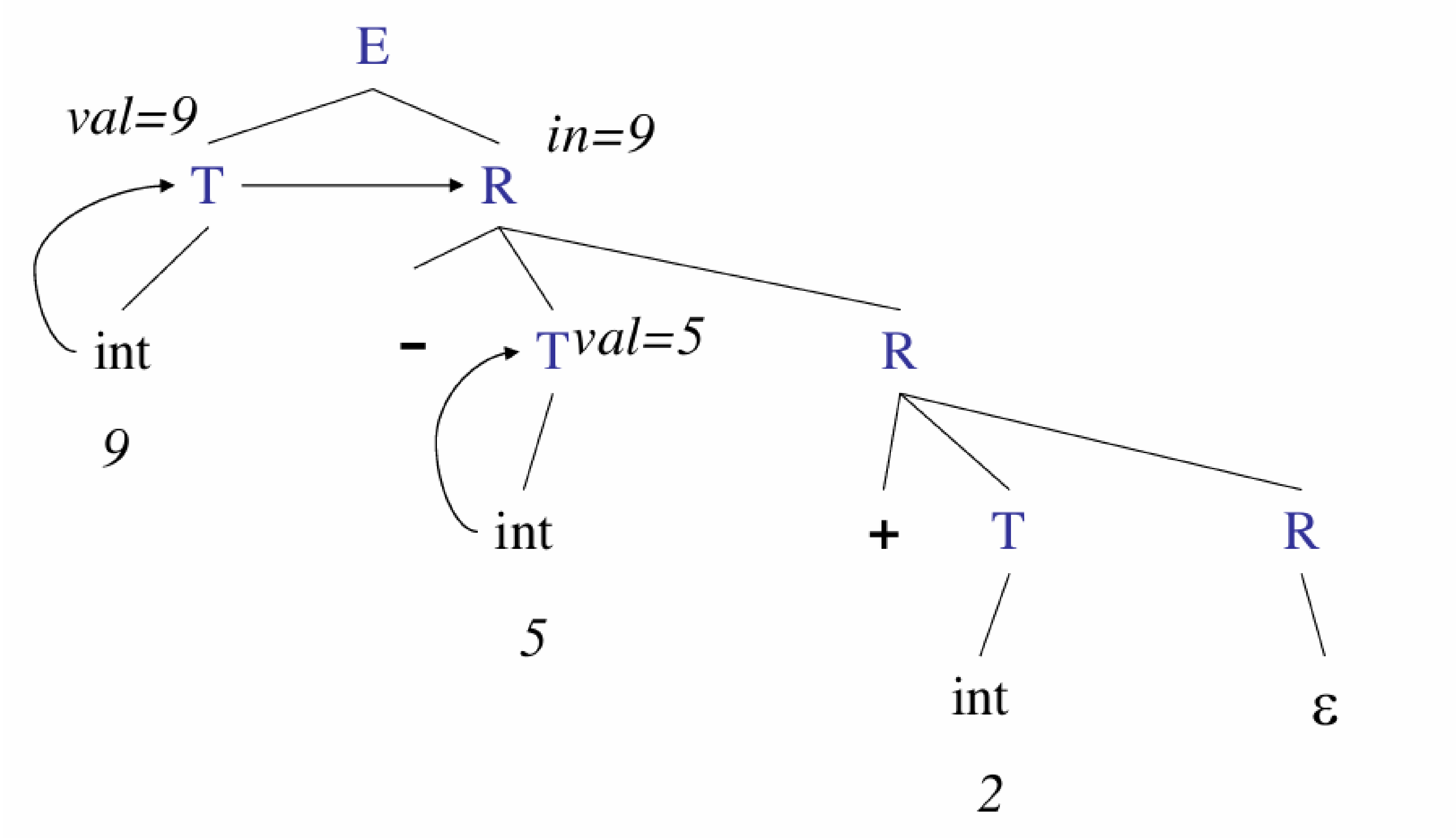
input: 9 - 5 + 2



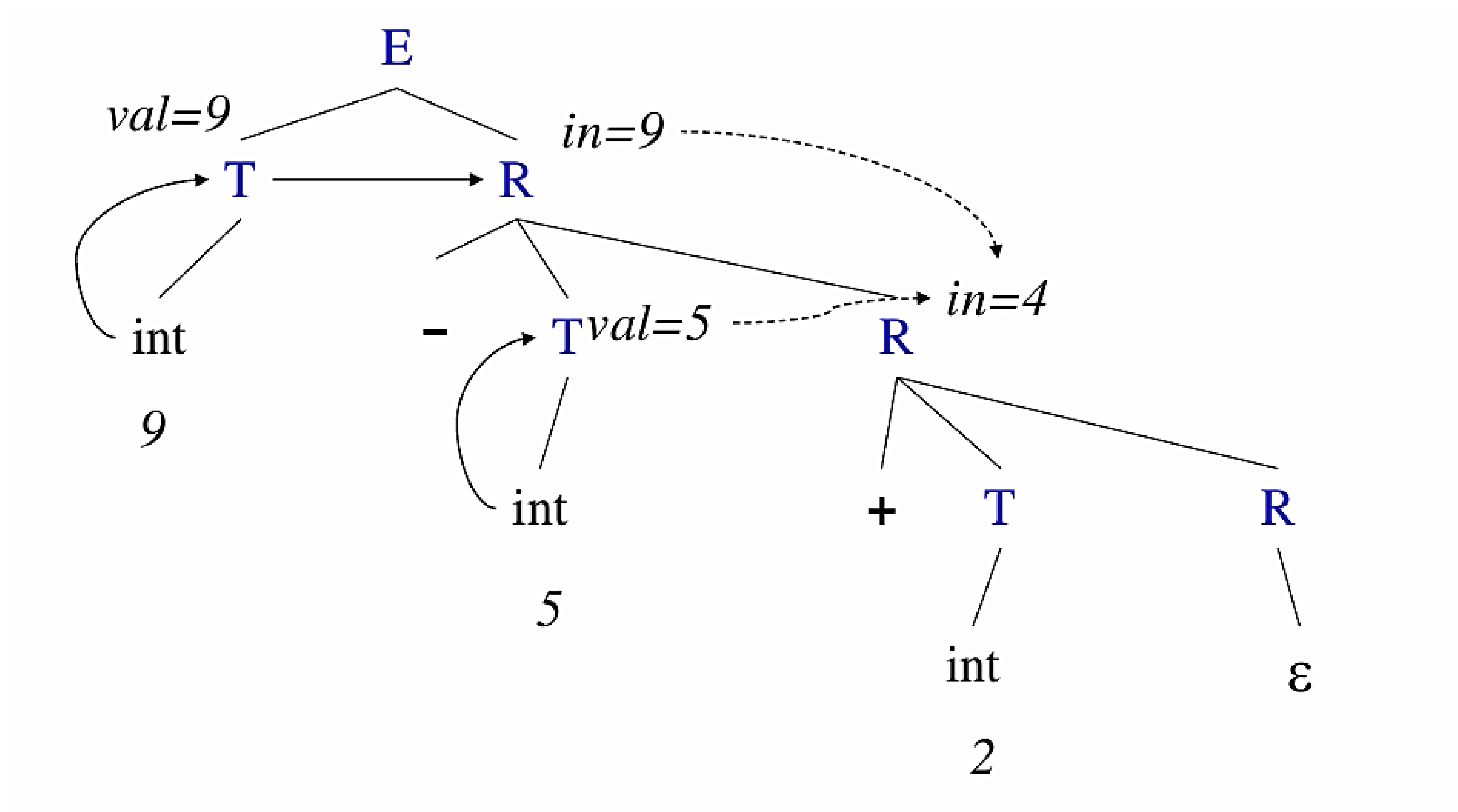
input: 9 - 5 + 2



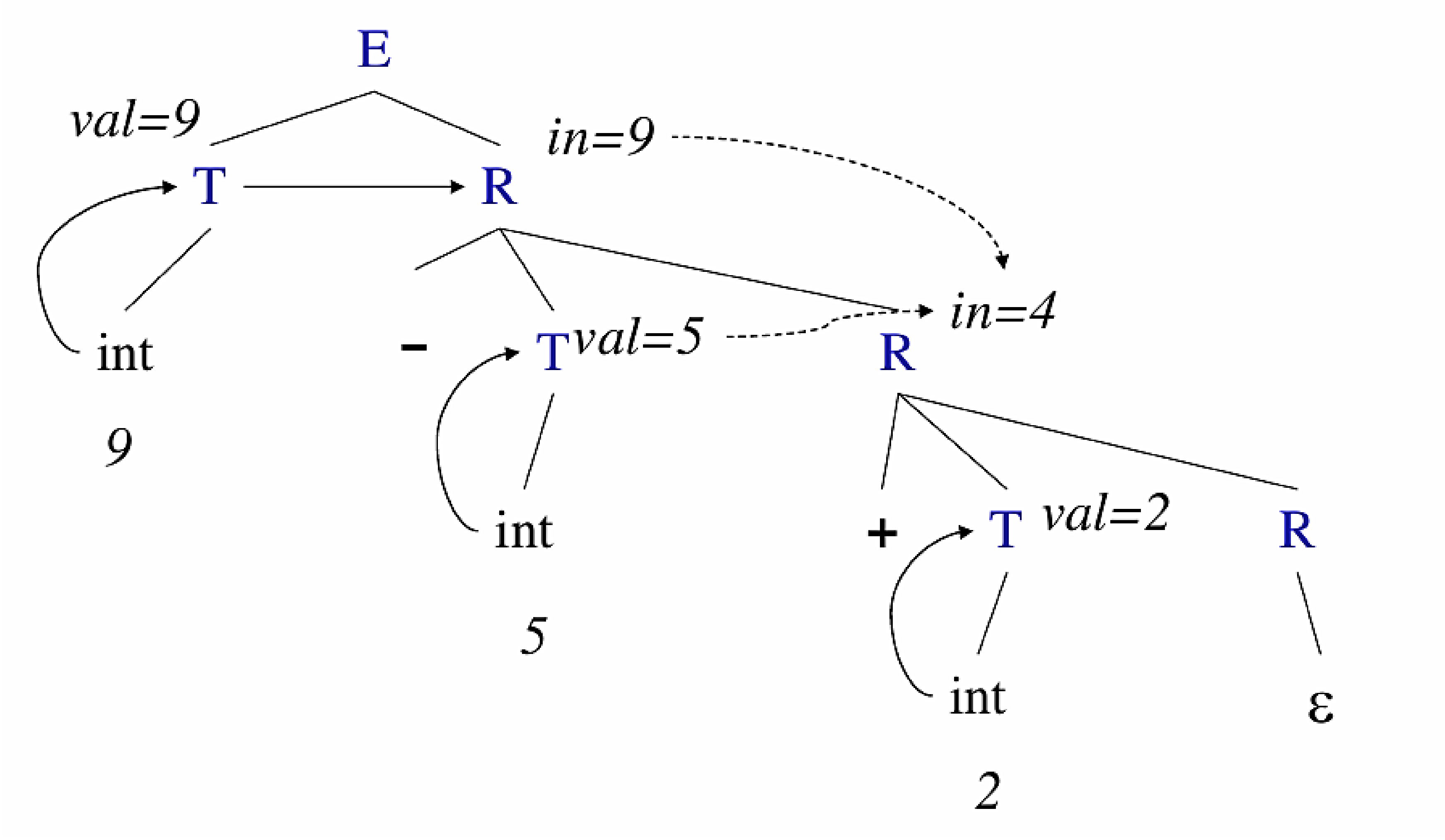
input: 9 - 5 + 2



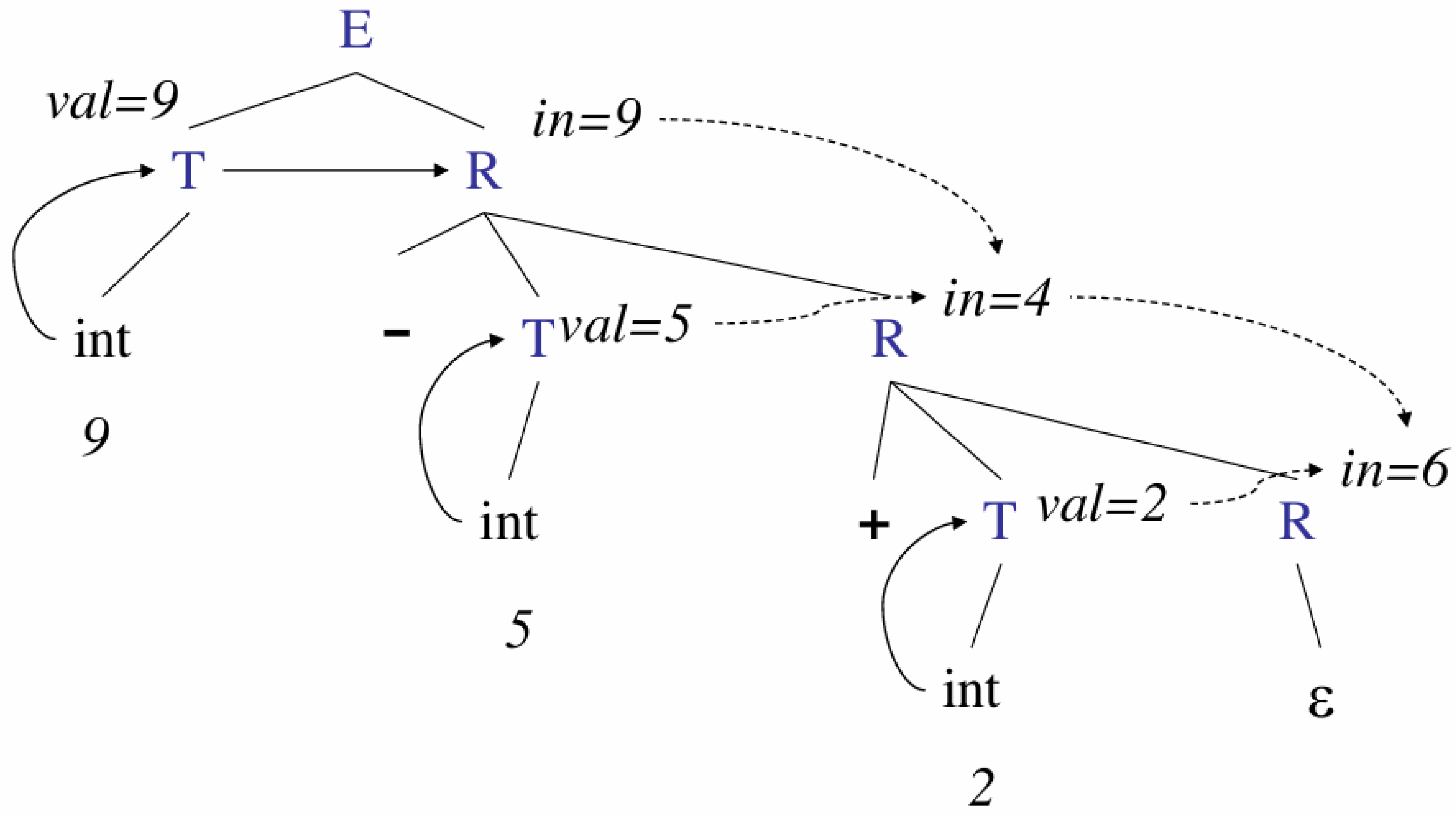
input: 9 - 5 + 2



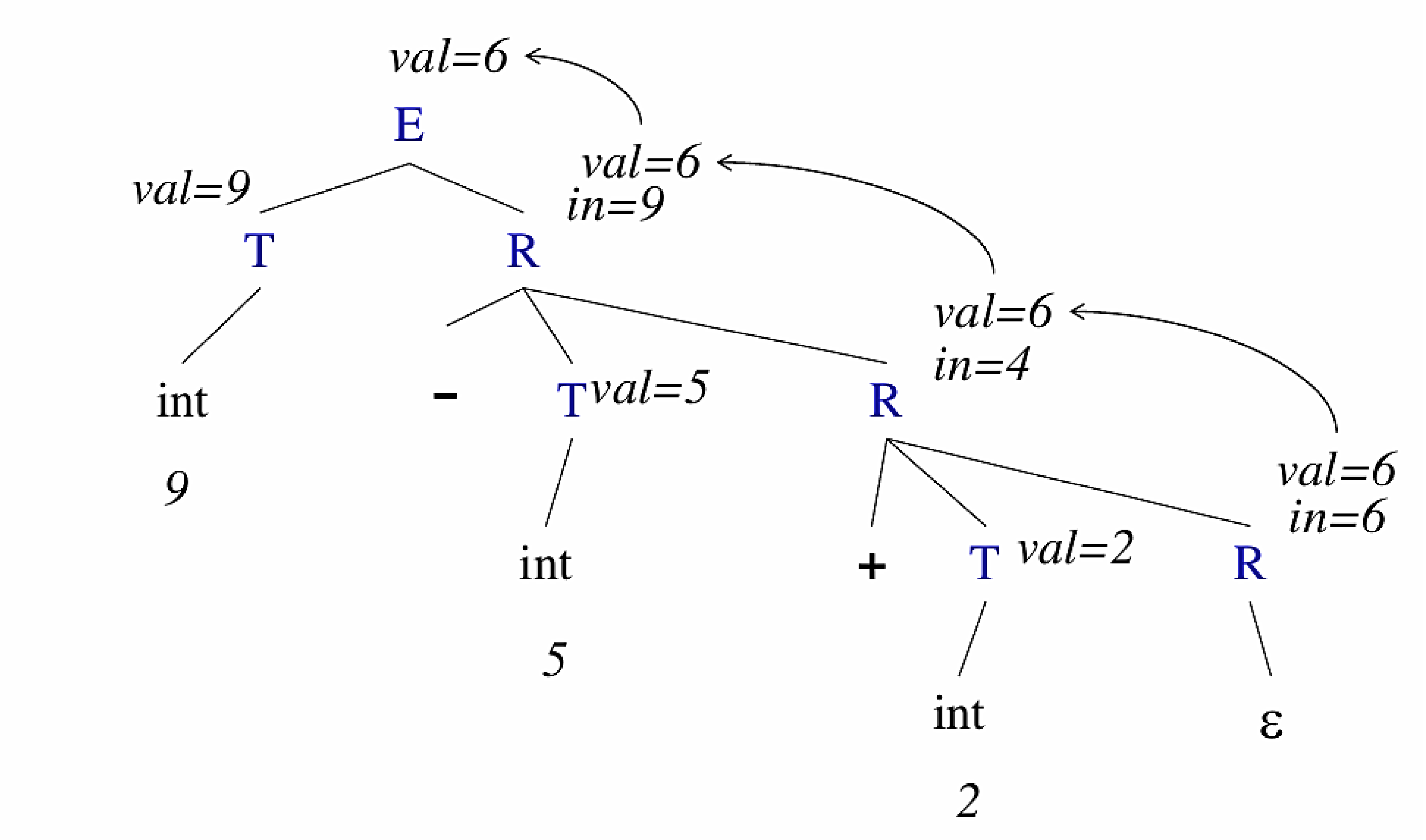
input: 9 - 5 + 2



input: 9 - 5 + 2

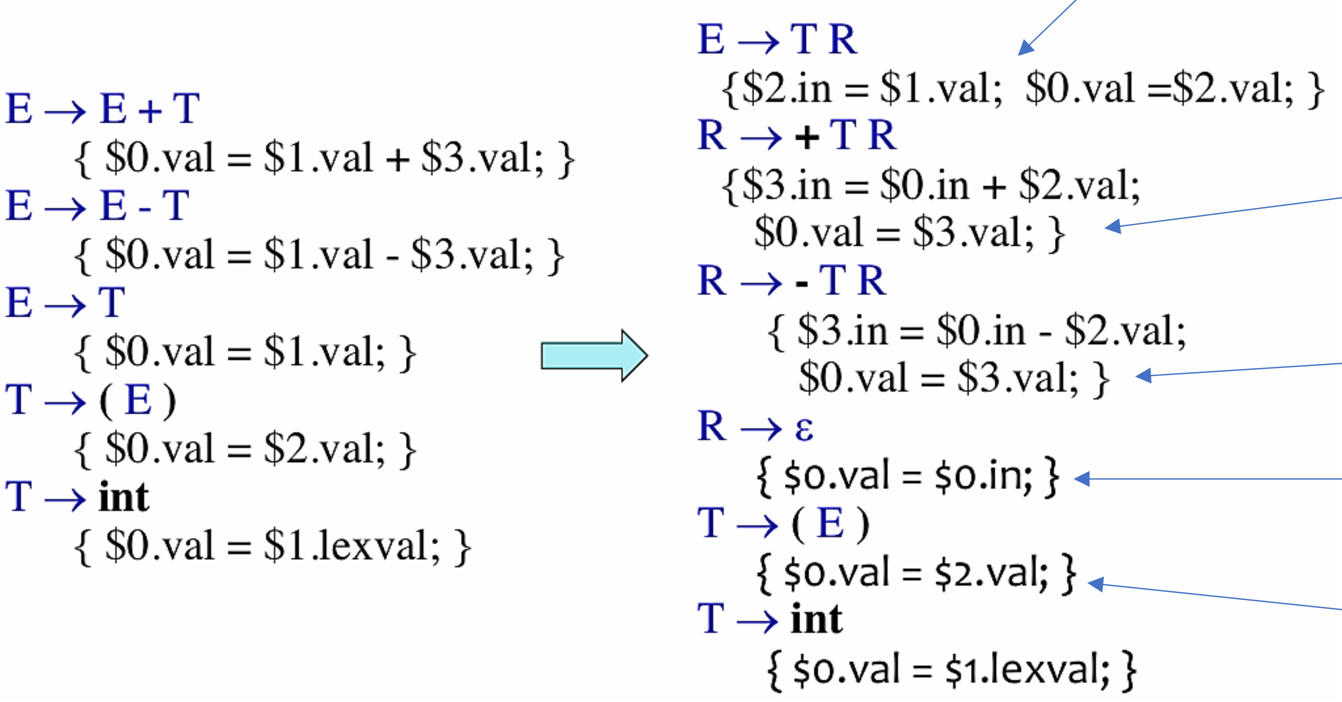


input: 9 - 5 + 2


val=6
E
val=9
T
val=6
in=9
R
int
9
-
T val=5
int
5
R
val=6
in=4
+
T val=2
int
2
R
val=6
in=6
ε

Top-down translation example

- SDT for the LL(1) grammar:



E → E + T
{ $0.val = $1.val + $3.val; }
E → E - T
{ $0.val = $1.val - $3.val; }
E → T
{ $0.val = $1.val; }
T → (E)
{ $0.val = $2.val; }
T → **int**
{ $0.val = $1.lexval; }

E → T R
{$2.in = $1.val; $0.val =$2.val; }
R → **+** T R
{$3.in = $0.in + $2.val;
$0.val = $3.val; }
R → **-** T R
{ $3.in = $0.in - $2.val;
$0.val = $3.val; }
R → ε
{ $o.val = $o.in; }
T → (E)
{ $o.val = $2.val; }
T → **int**
{ $o.val = $1.lexval; }

Pass value from **`T`** to **`R`** as input (inherited), then take final result from **`R`.**

Add T.val to current running total (R.in), pass result along.

Subtract **`T.val`** from current total, pass it forward.

When recursion ends, return current total.

Value of **`T`** is the value of the inner expression.

Value of **`T`** is the numeric value of the integer token.