Syntax Directed Translation

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Syntax directed Translation

- Models for translation from parse trees into assembly/machine code
- Representation of translations
  - Attribute Grammars (semantic actions for CFGs)
  - Tree Matching Code Generators
  - Tree Parsing Code Generators

Attribute Grammars

- Syntax-directed translation uses a grammar to produce code (or any other “semantics”)
- Consider this technique to be a generalization of a CFG definition
- Each grammar symbol is associated with an attribute
- An attribute can be anything: a string, a number, a tree, any kind of record or object

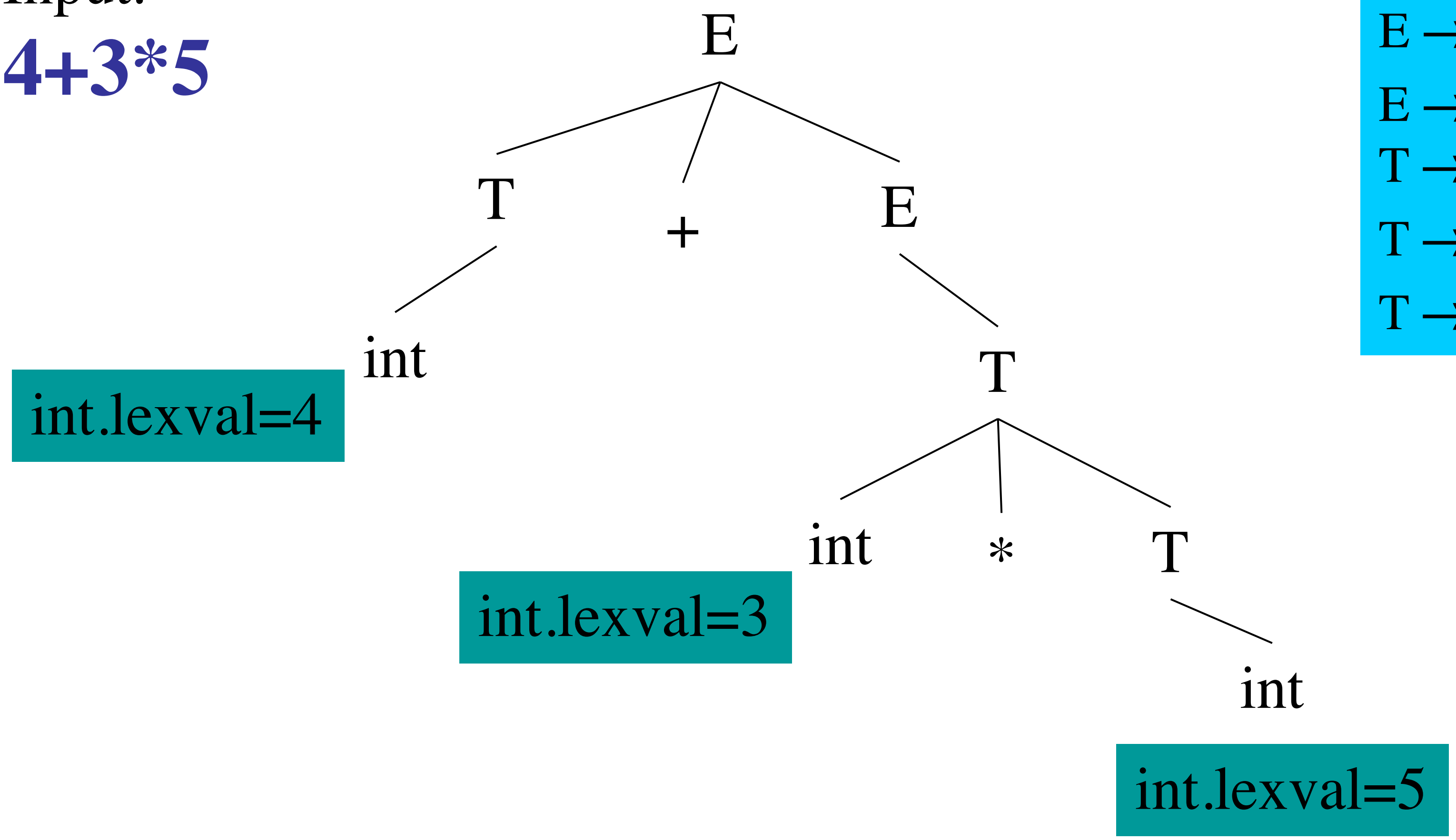
Attribute Grammars

- A CFG can be viewed as a (finite) representation of a function that relates strings to parse trees
- Similarly, an attribute grammar is a way of relating strings with "meanings"
- Since this relation is syntax-directed, we associate each CFG rule with a semantic (rules to build an abstract syntax tree)
- In other words, attribute grammars are a method to *decorate* or *annotate* the parse tree

Expr concrete syntax tree

Input:

4+3*5



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

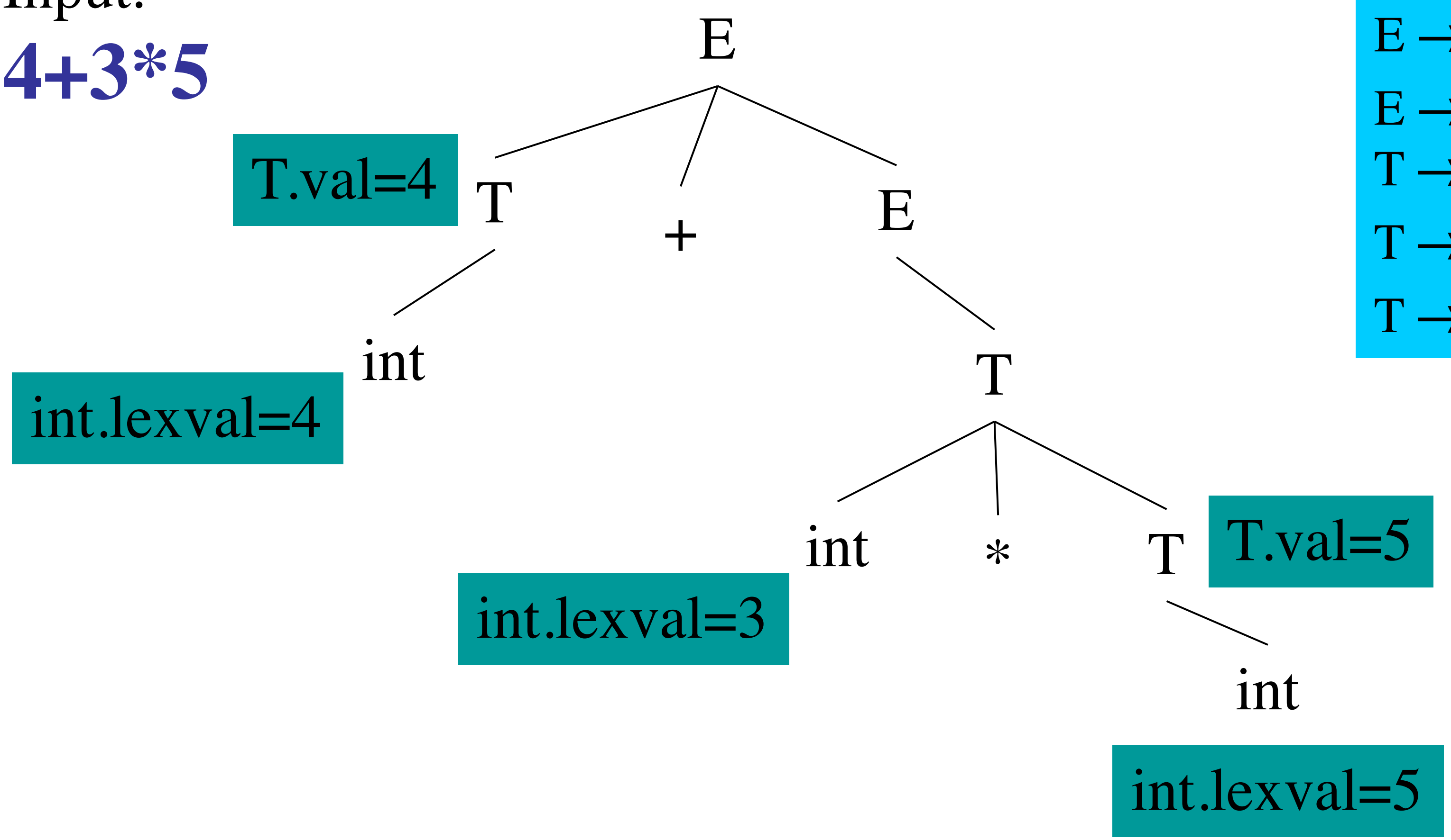
$T \rightarrow \text{int} * T$

$T \rightarrow (E)$

Expr concrete syntax tree

Input:

4+3*5



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

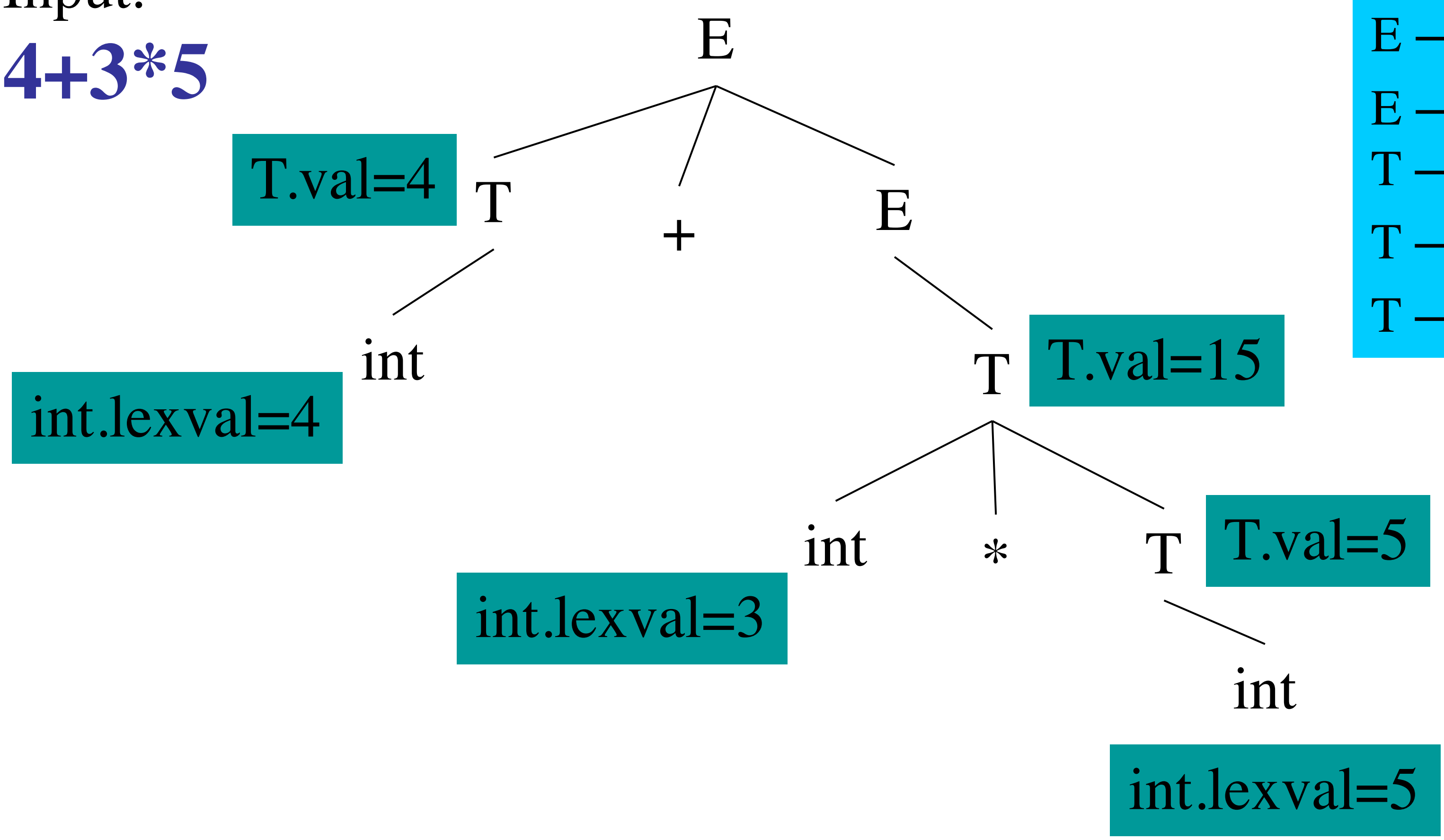
$T \rightarrow \text{int} * T$

$T \rightarrow (E)$

Expr concrete syntax tree

Input:

4+3*5



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

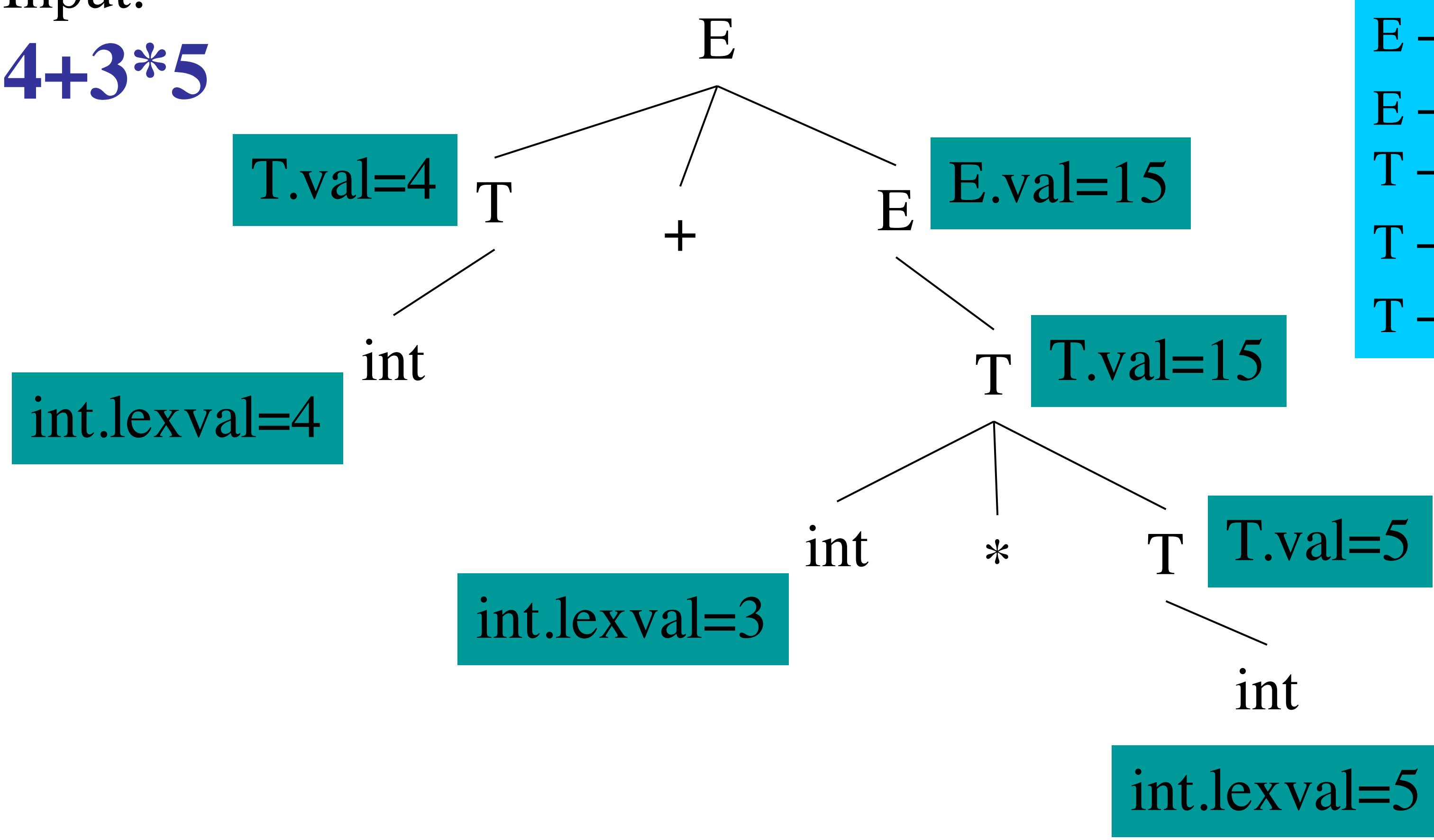
$T \rightarrow \text{int} * T$

$T \rightarrow (E)$

Expr concrete syntax tree

Input:

4+3*5



E → T + E

E → T

T → int

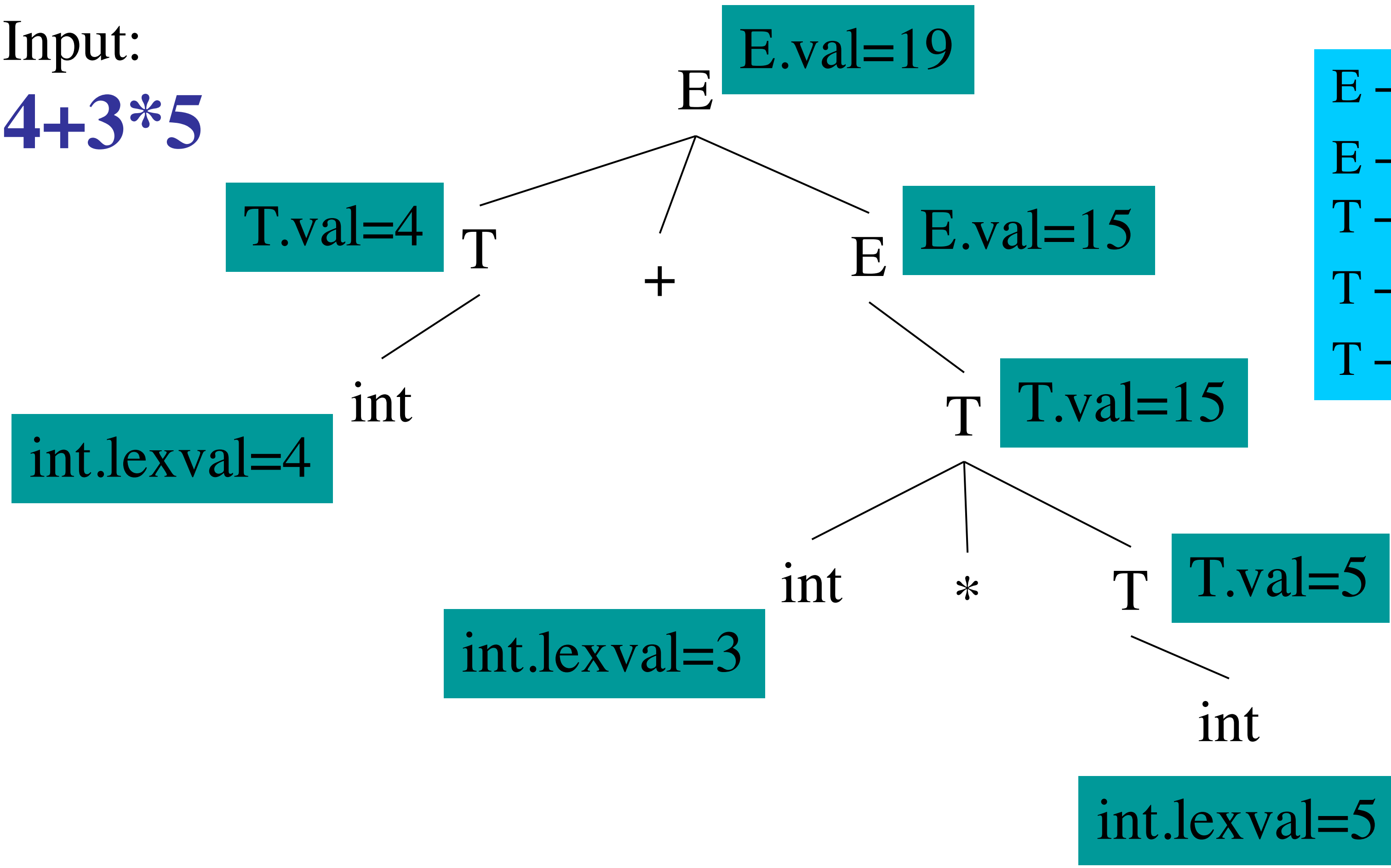
T → int * T

T → (E)

Expr concrete syntax tree

Input:

4+3*5



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow int$

$T \rightarrow int * T$

$T \rightarrow (E)$

Syntax directed definition

$T \rightarrow$ **int**

{ $0.val = $1.lexval; } ➔ In yacc: { $$ = $1 }

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

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

$E \rightarrow T$

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

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

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

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

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

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**

Inherited Attributes

- Synthesized attributes may not be sufficient for all cases that might arise for semantic checking and code generation
- Consider the (sub)grammar:

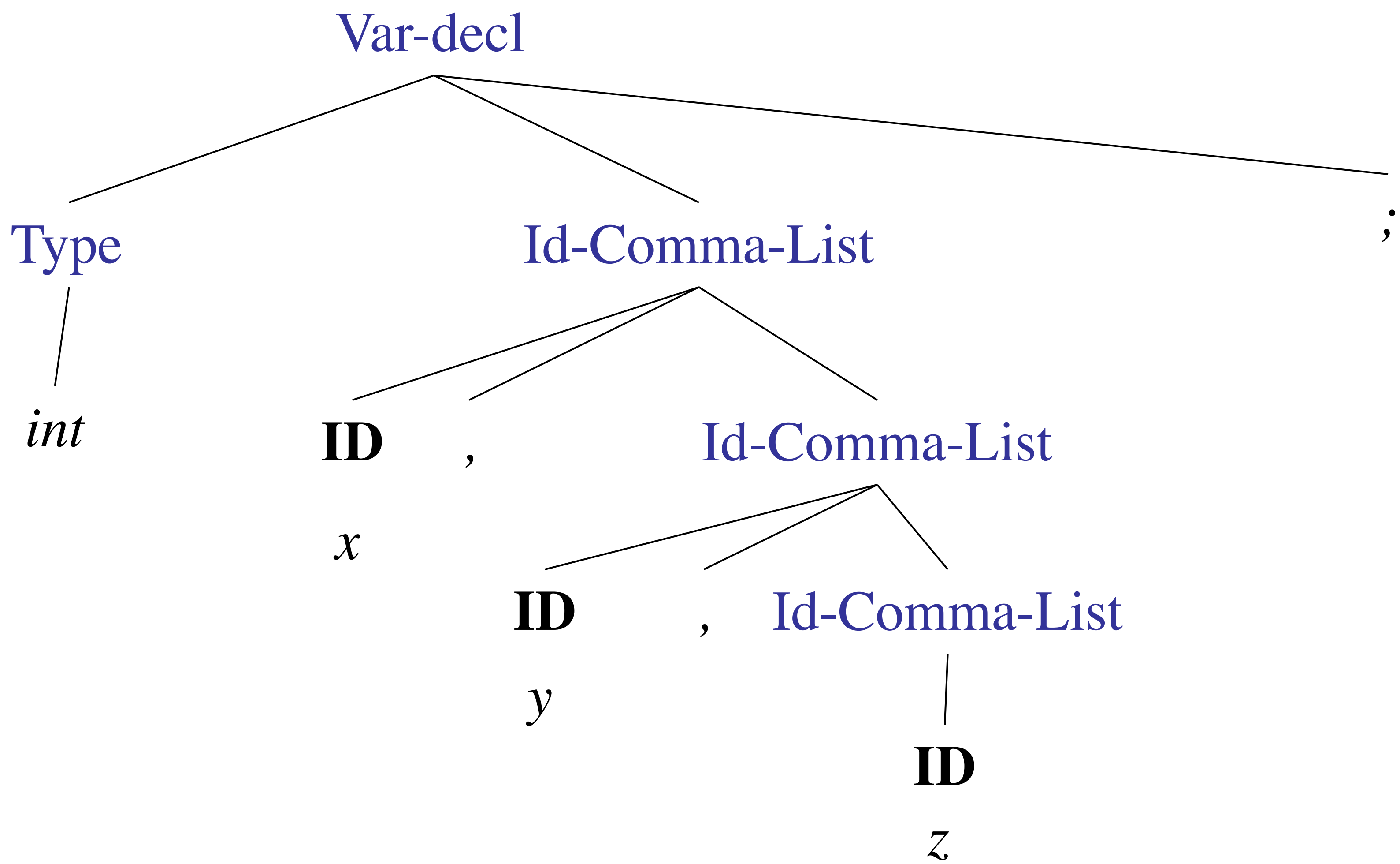
  Var-decl $\rightarrow$ Type Id-comma-list **;**

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

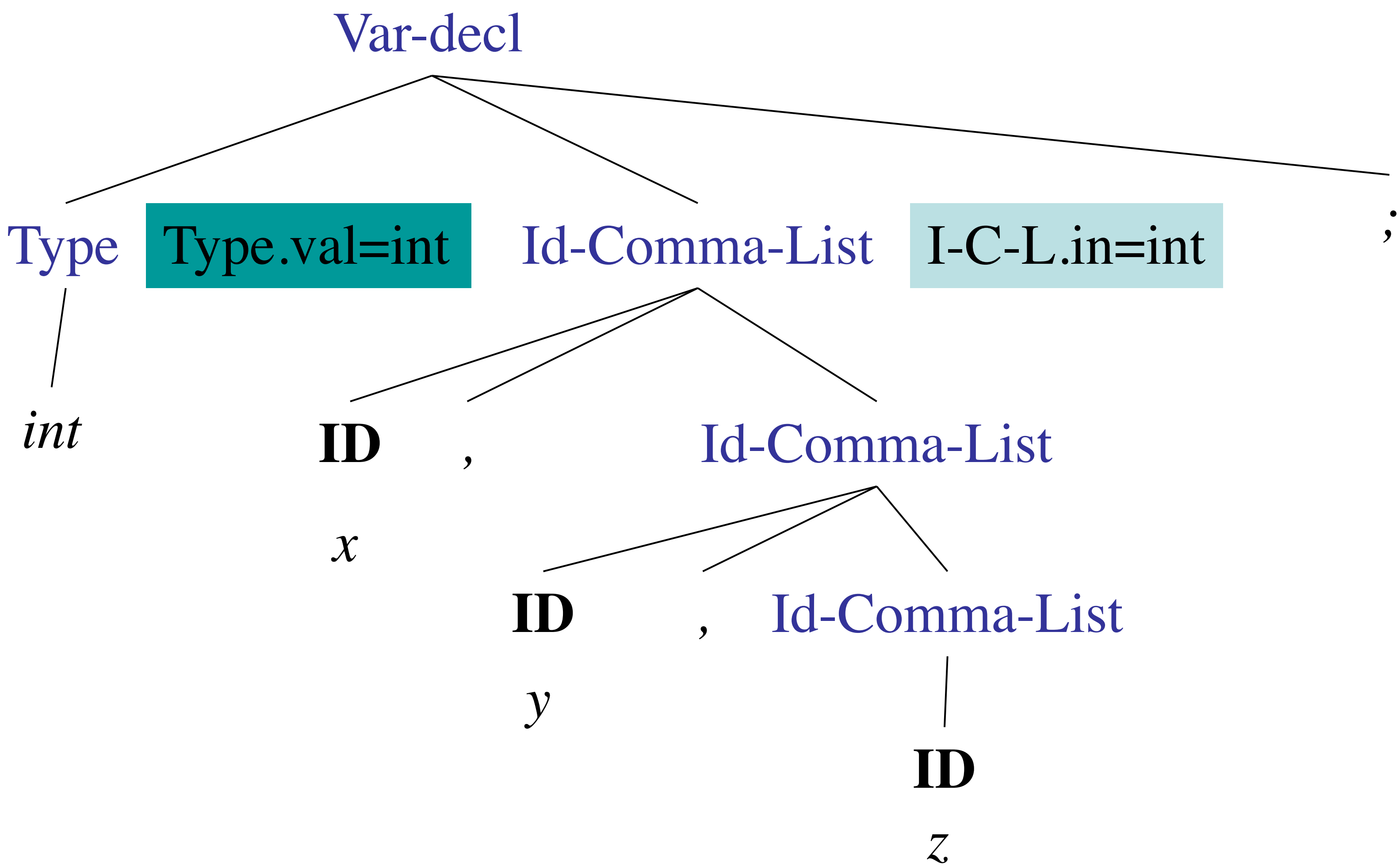
  Id-comma-list $\rightarrow$ **ID**

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

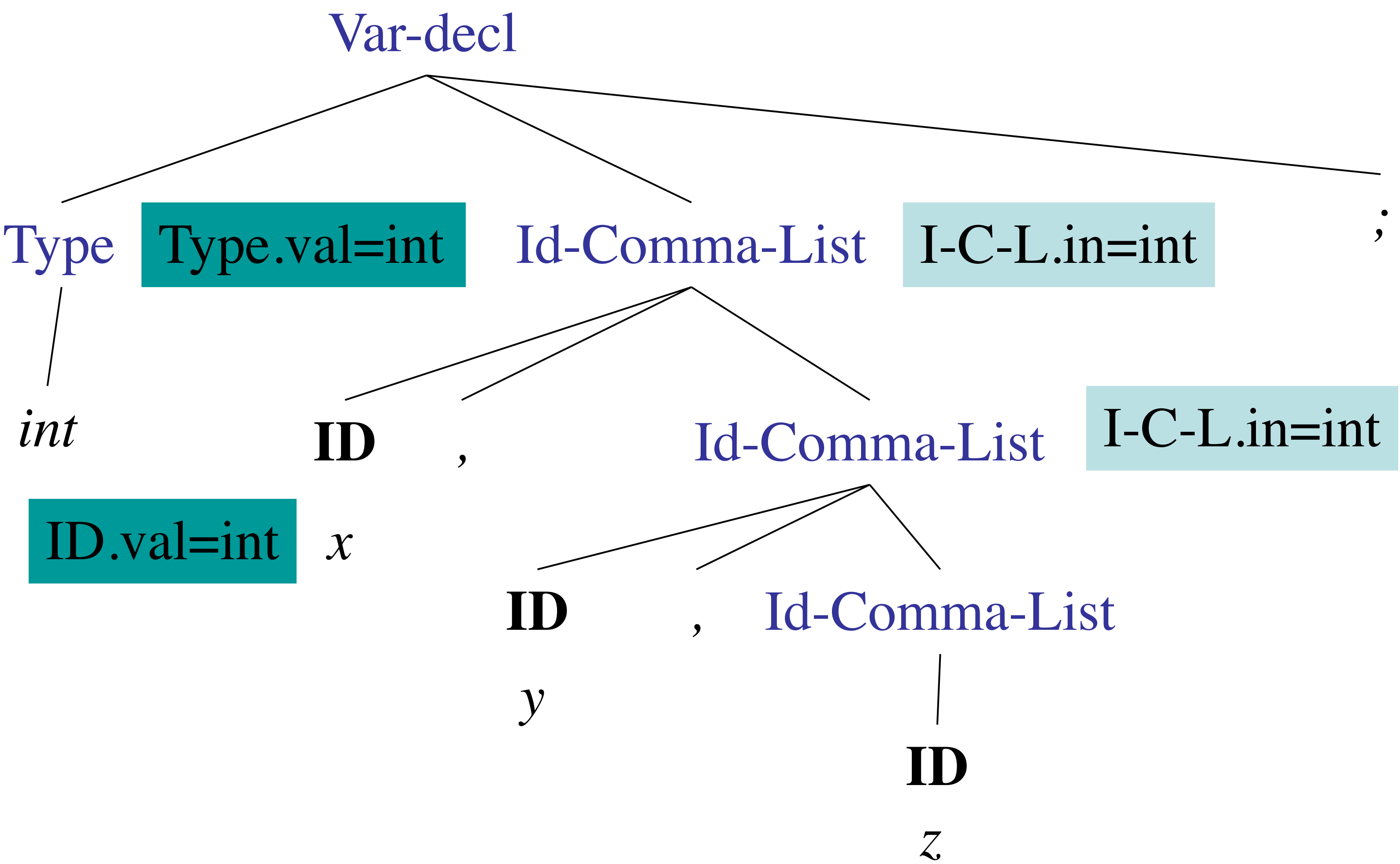
Example input: *int x, y, z ;*



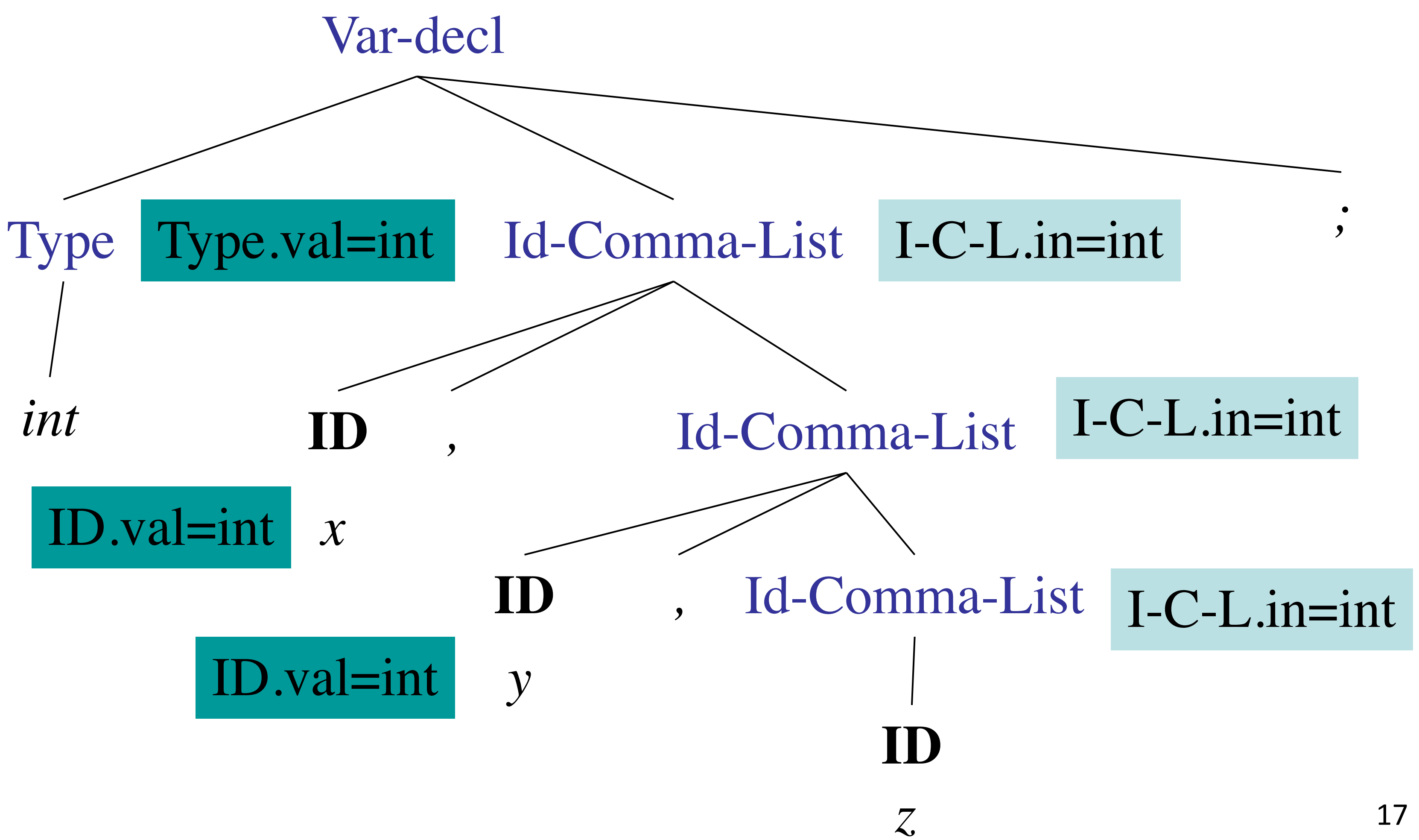
Example input: *int x, y, z ;*



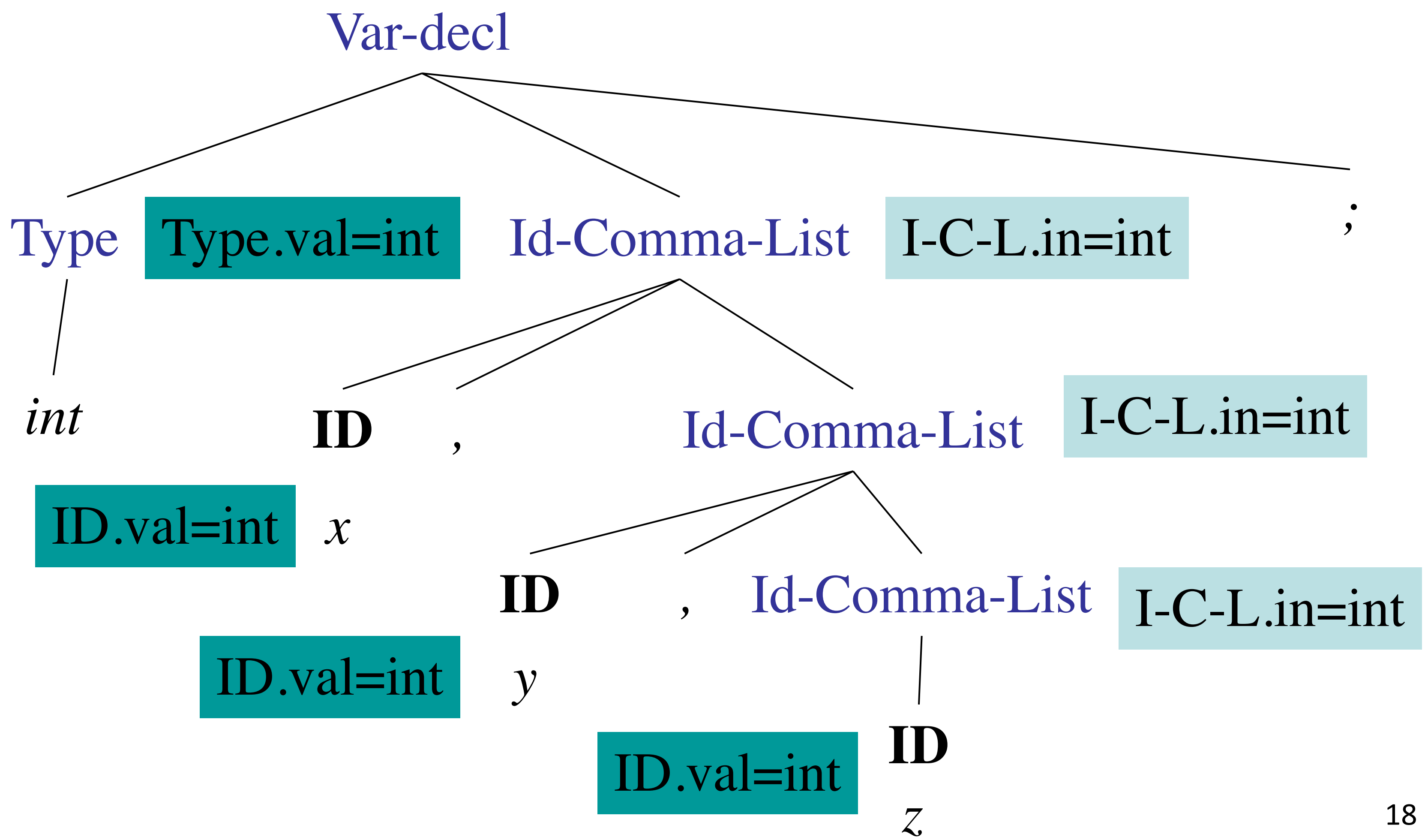
Example input: *int x, y, z ;*



Example input: *int x, y, z ;*



Example input: *int x, y, z ;*



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
- *Id-Comma-List* takes its attribute from its left sibling *Type* (or from its parent *Id-Comma-list*)

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; }

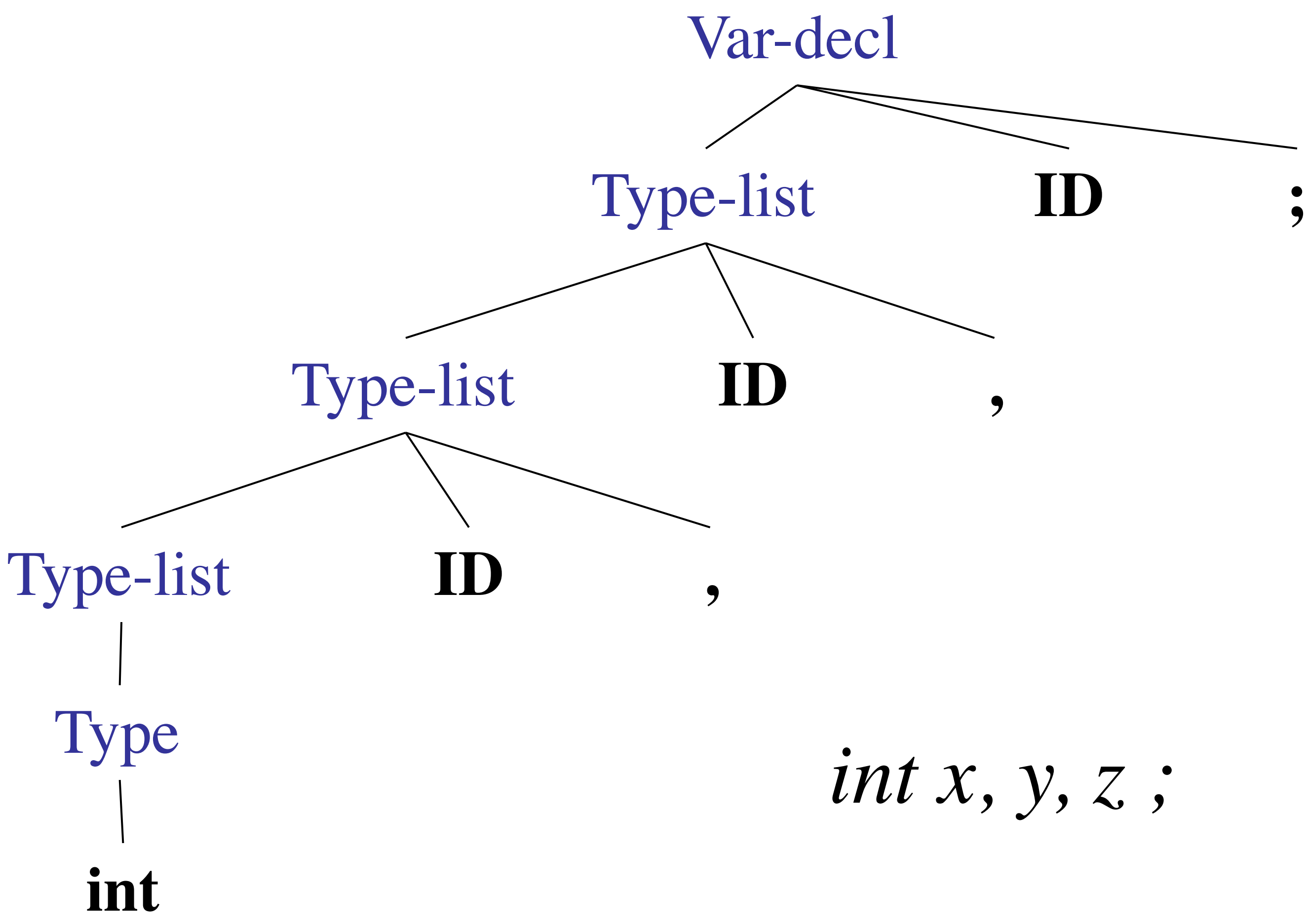
Id-comma-list → **ID ,** Id-comma-list

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

Inherited Attributes

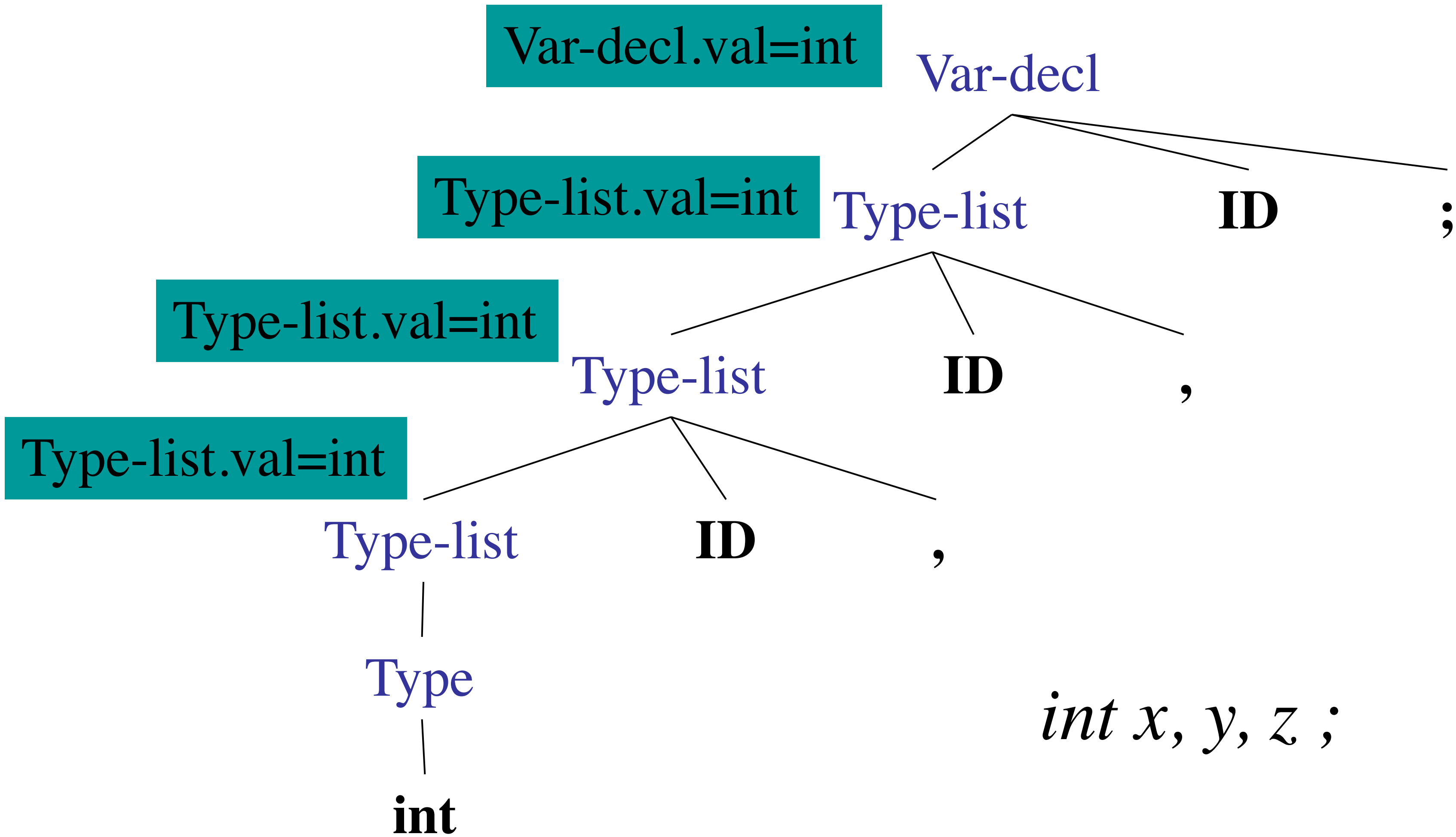
- **Inherited attributes** are attributes that are computed at a node based on attributes from siblings or the parent
- Typically we combine synthesized attributes and inherited attributes
- It is possible to convert the grammar into a form that *only* uses synthesized attributes

Removing Inherited Attributes



int x, y, z ;

Removing Inherited Attributes



int x, y, z ;

Removing inherited attributes

Var-decl $\rightarrow$ Type-List **ID ;**

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

Type-list $\rightarrow$ Type-list **ID ,**

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

Type-list $\rightarrow$ Type

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

Type $\rightarrow$ **int**

{ $0.val = int; }

| **bool**

{ $0.val = bool; }

Direction of inherited attributes

- Consider the syntax directed defns:

  $A \rightarrow L\ M$

  { \$1.in = \$0.in; \$2.in = \$1.val; \$0.val = \$2.val; }

  $A \rightarrow Q\ R$

  { \$2.in = \$0.in; \$1.in = \$2.val; \$0.val = \$1.val; }

- Problematic definition: \$1.in = \$2.val
- Difference between incremental processing vs. using the completed parse tree

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
- Some inherited attributes cannot be used in conjunction with incremental processing

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 $j=1 \ldots n$, each inherited attribute of X_j depends on:
  - The attributes of $X_1 \ldots X_{j-1}$
  - 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 which is used.
- Two important classes of SDTs:

1. LR parser, syntax directed definition is S-attributed
2. LL parser, syntax directed definition is L-attributed

Syntax-directed defns

- 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

$E \to E + T$

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

$E \to E - T$

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

$T \to$ **int**

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

$E \to T$

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

$T \to$ **(** E **)**

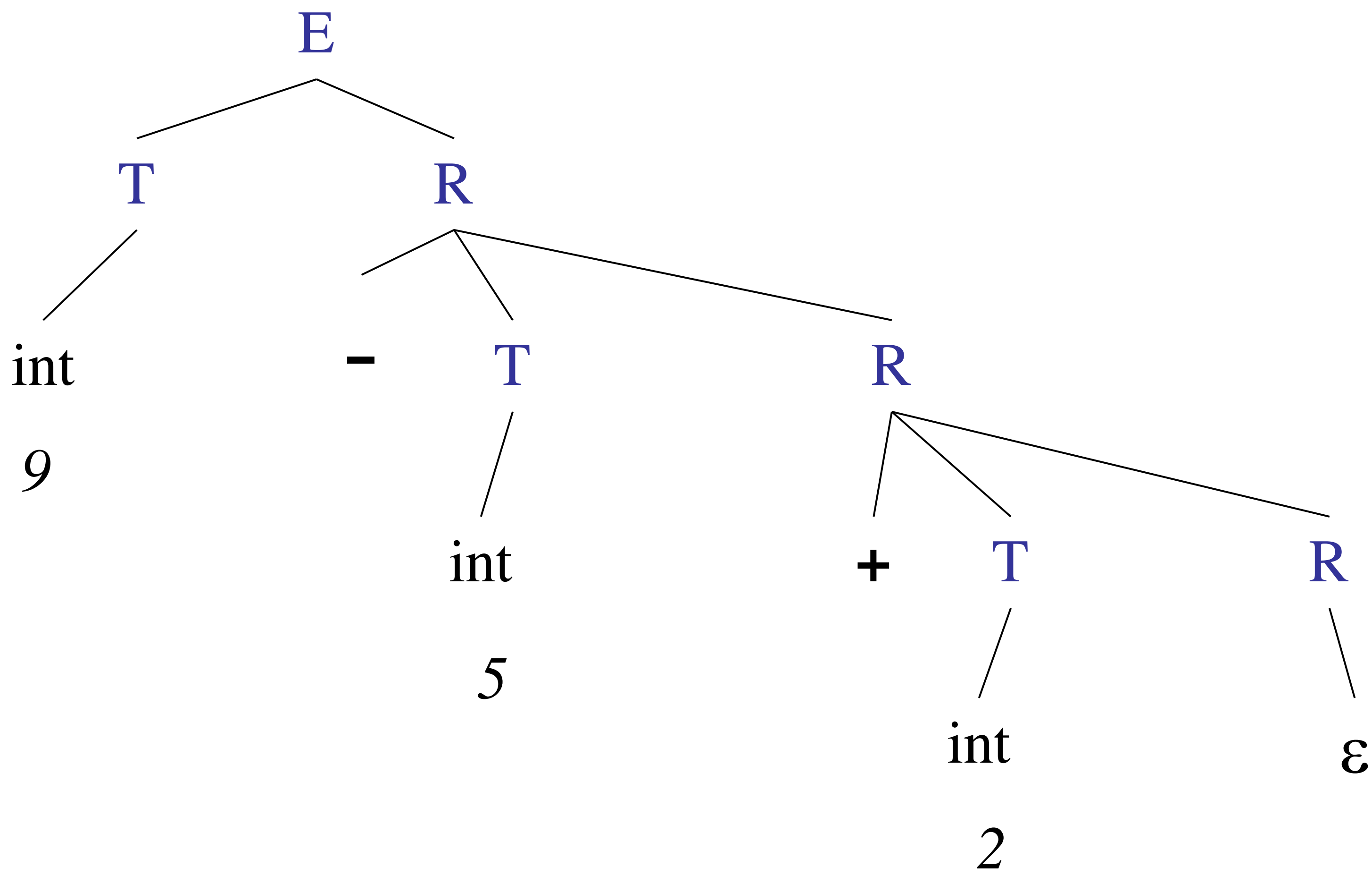
{ $0.val = $2.val; }

Top-down translation
example

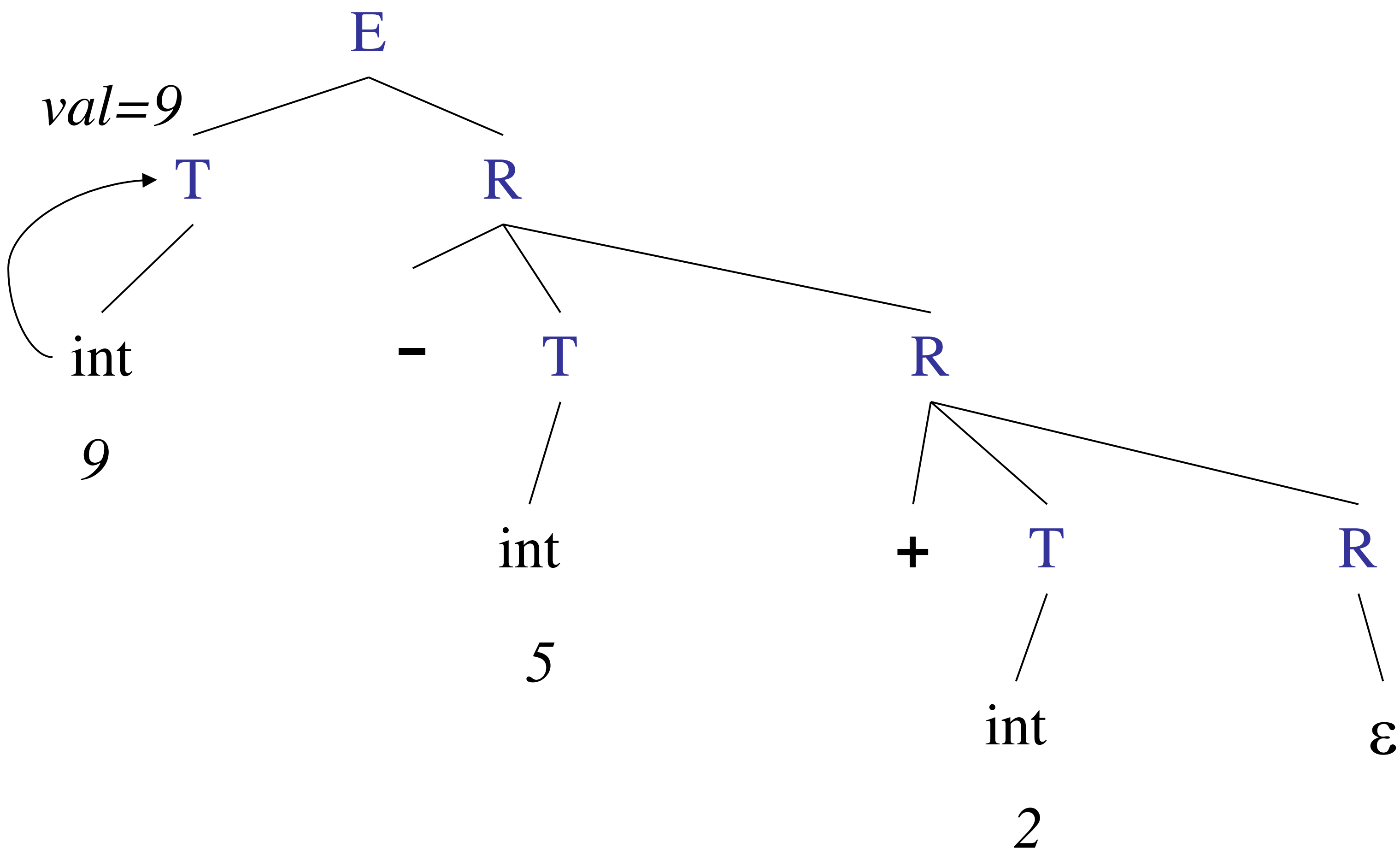
Remove left recursion

$$
\begin{aligned}
E &\to E + T \\
E &\to E - T \\
E &\to T \\
T &\to (\ E\ ) \\
T &\to \textbf{int}
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
E &\to T\ R \\
R &\to \textbf{+}\ T\ R \\
R &\to \textbf{-}\ T\ R \\
R &\to \varepsilon \\
T &\to (\ E\ ) \\
T &\to \textbf{int}
\end{aligned}
$$

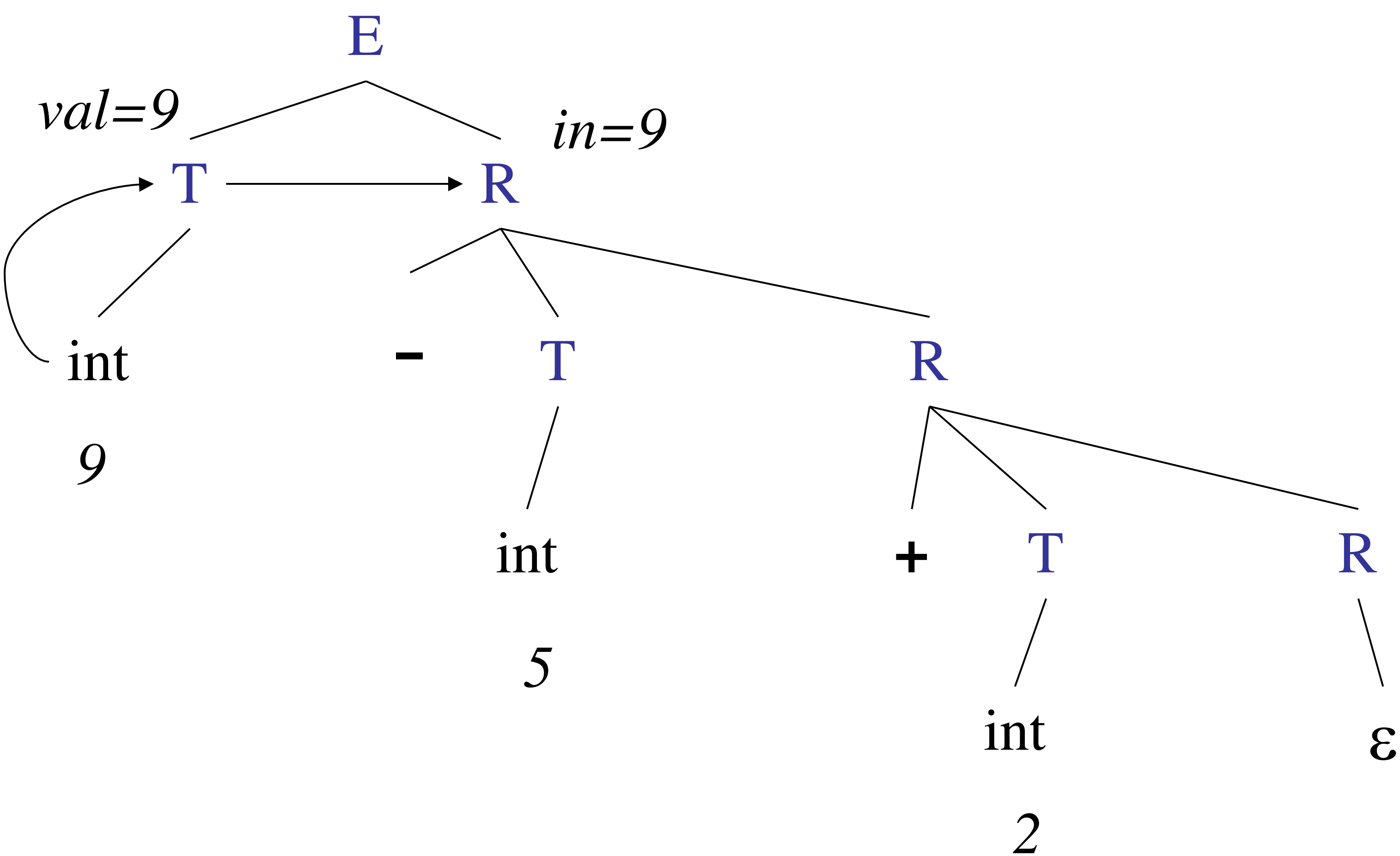
input: *9 - 5 + 2*



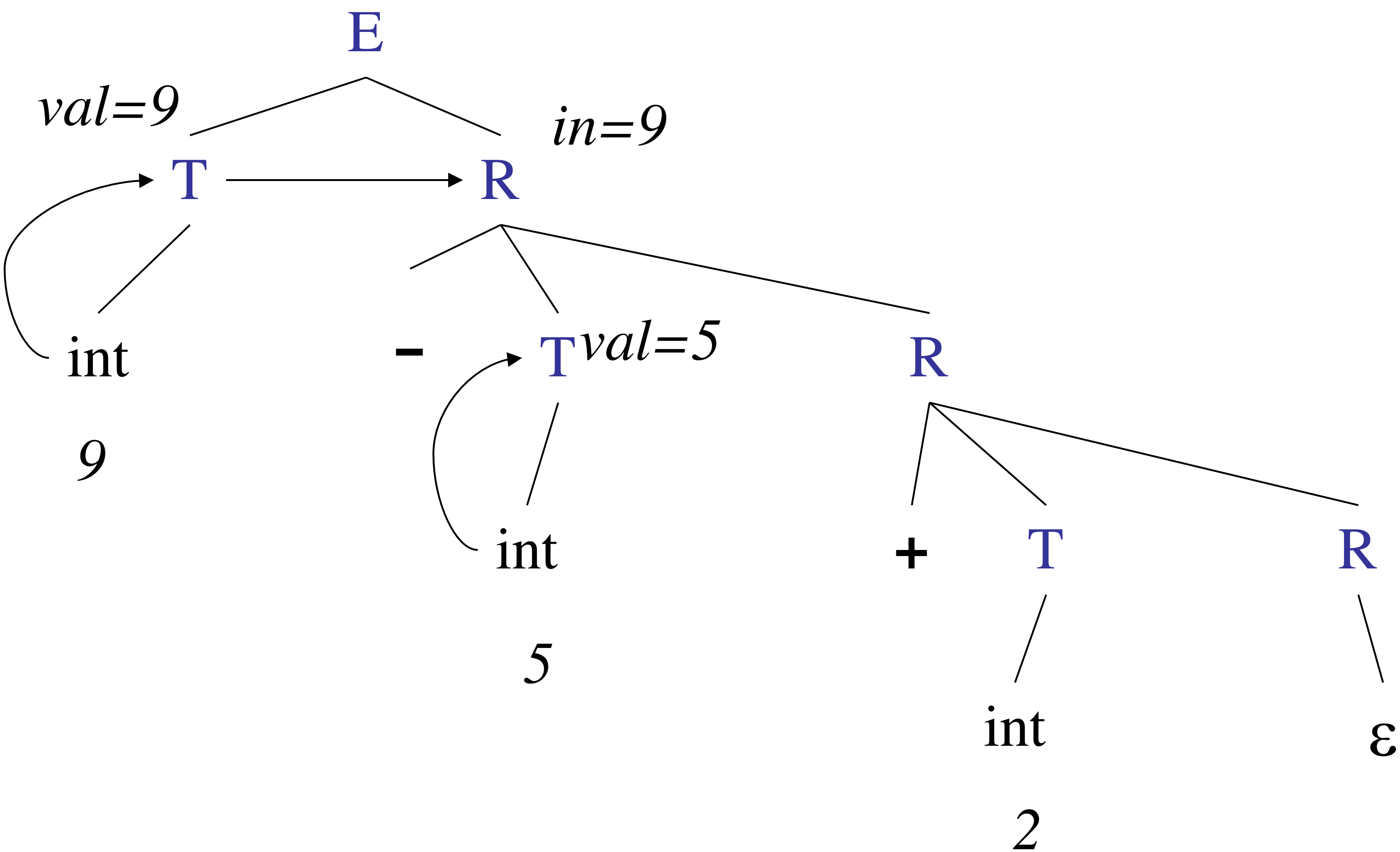
input: *9 - 5 + 2*



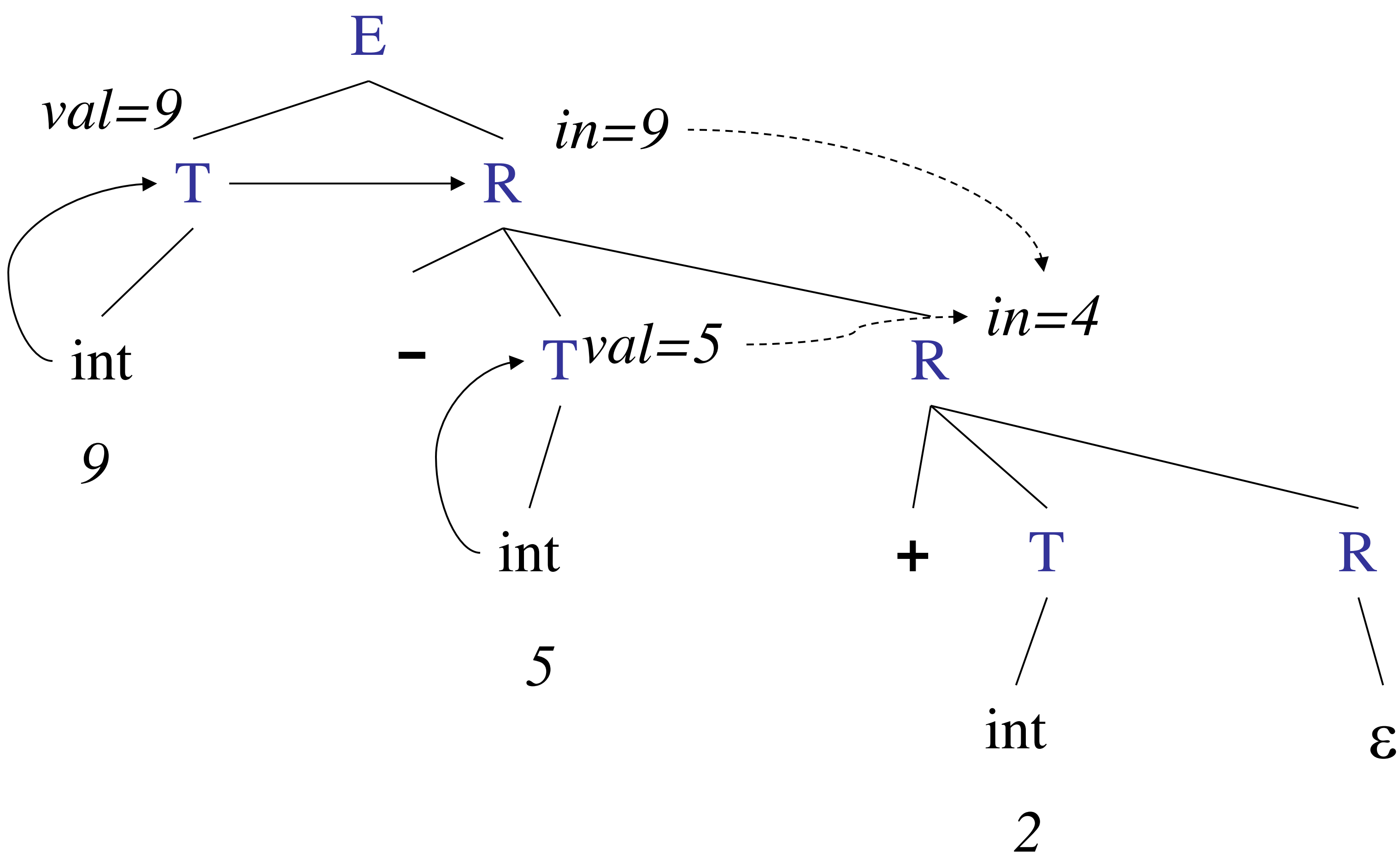
input: *9 - 5 + 2*



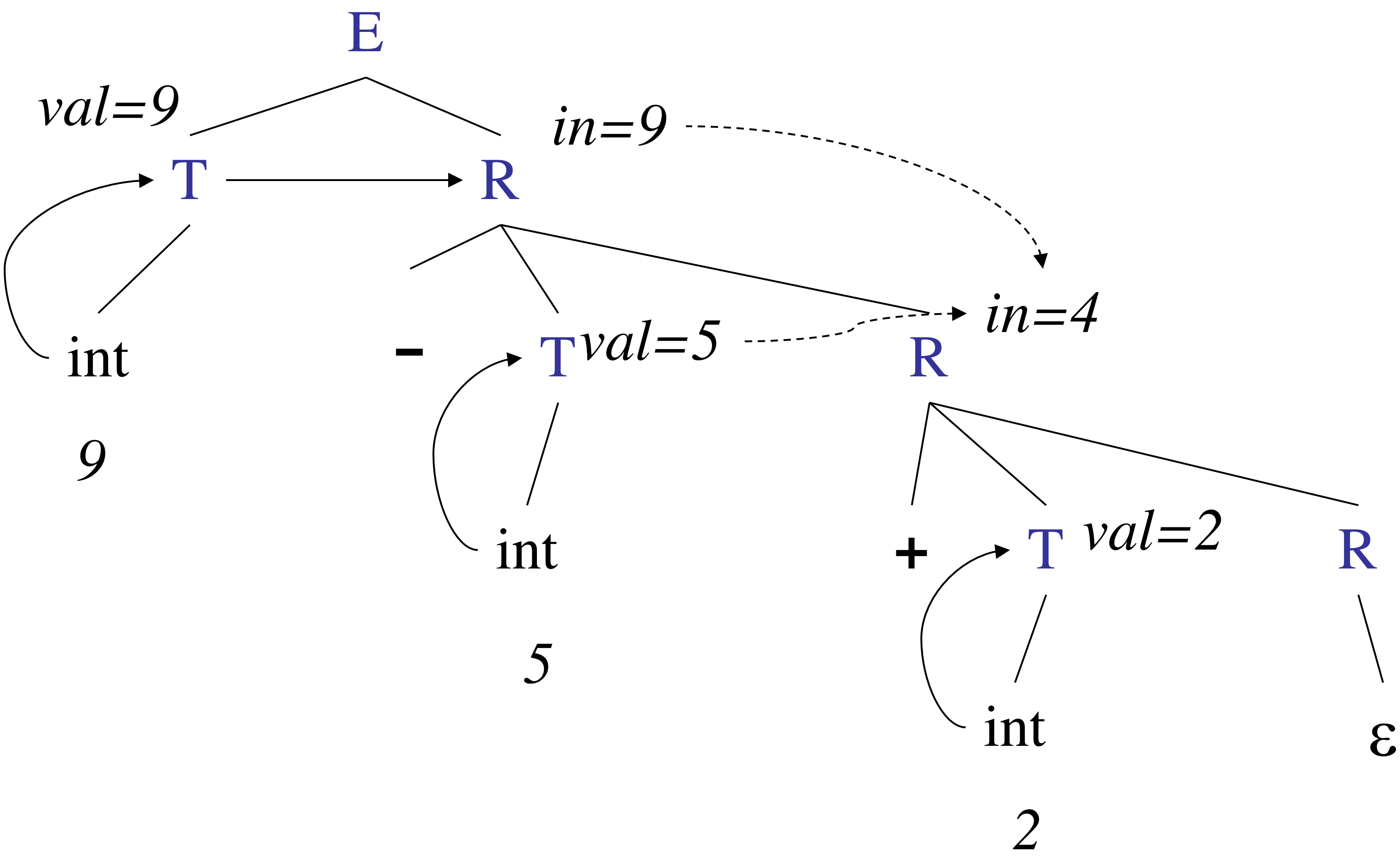
input: *9 - 5 + 2*



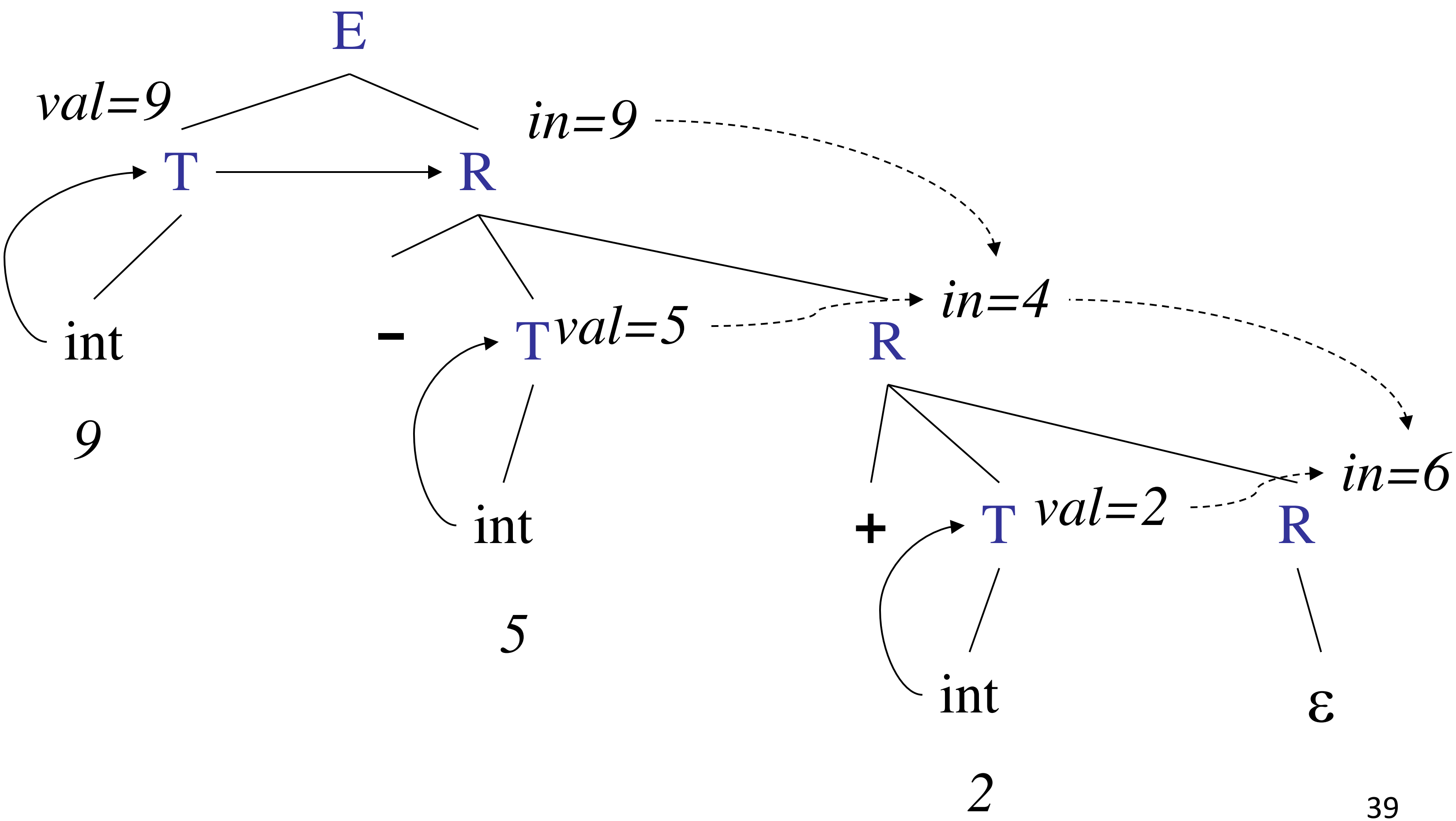
input: *9 - 5 + 2*



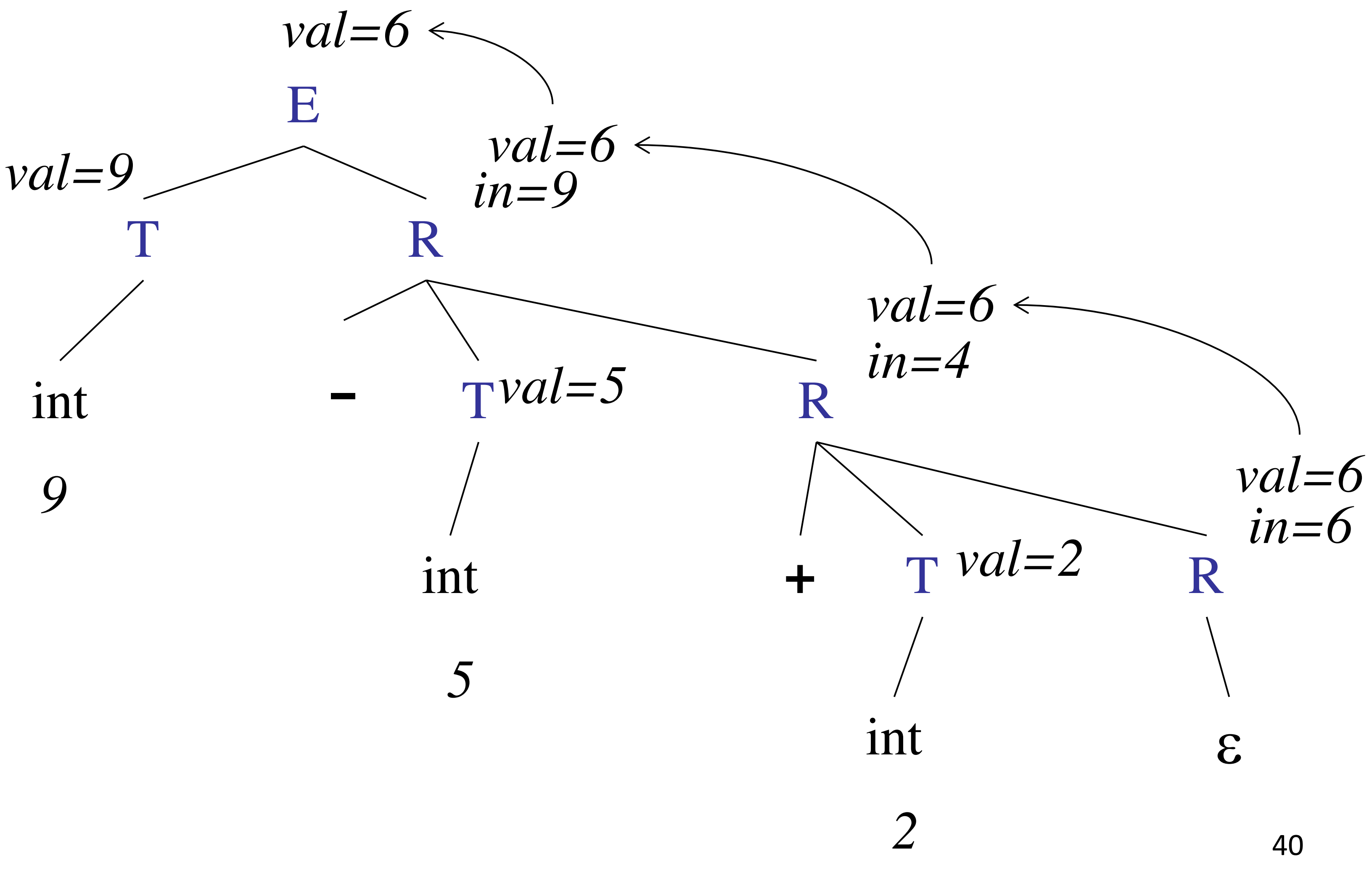
input: *9 - 5 + 2*



input: *9 - 5 + 2*



input: *9 - 5 + 2*



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

Top-down translation example

SDT for the LL(1) grammar:

$E \to E + T$

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

$E \to E - T$

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

$E \to T$

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

$T \to (E)$

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

$T \to$ **int**

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

$\Rightarrow$

$E \to T\ R$

{\$2.in = \$1.val; \$0.val =\$2.val; }

$R \to$ **+** $T\ R$

{\$3.in = \$0.in + \$2.val;
\$0.val = \$3.val; }

$R \to$ **-** $T\ R$

{ \$3.in = \$0.in - \$2.val;
\$0.val = \$3.val; }

$R \to \varepsilon$

{ \$0.val = \$0.in; }

$T \to (E)$

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

$T \to$ **int**

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

Dependencies and SDTs

- There can be circular definitions:

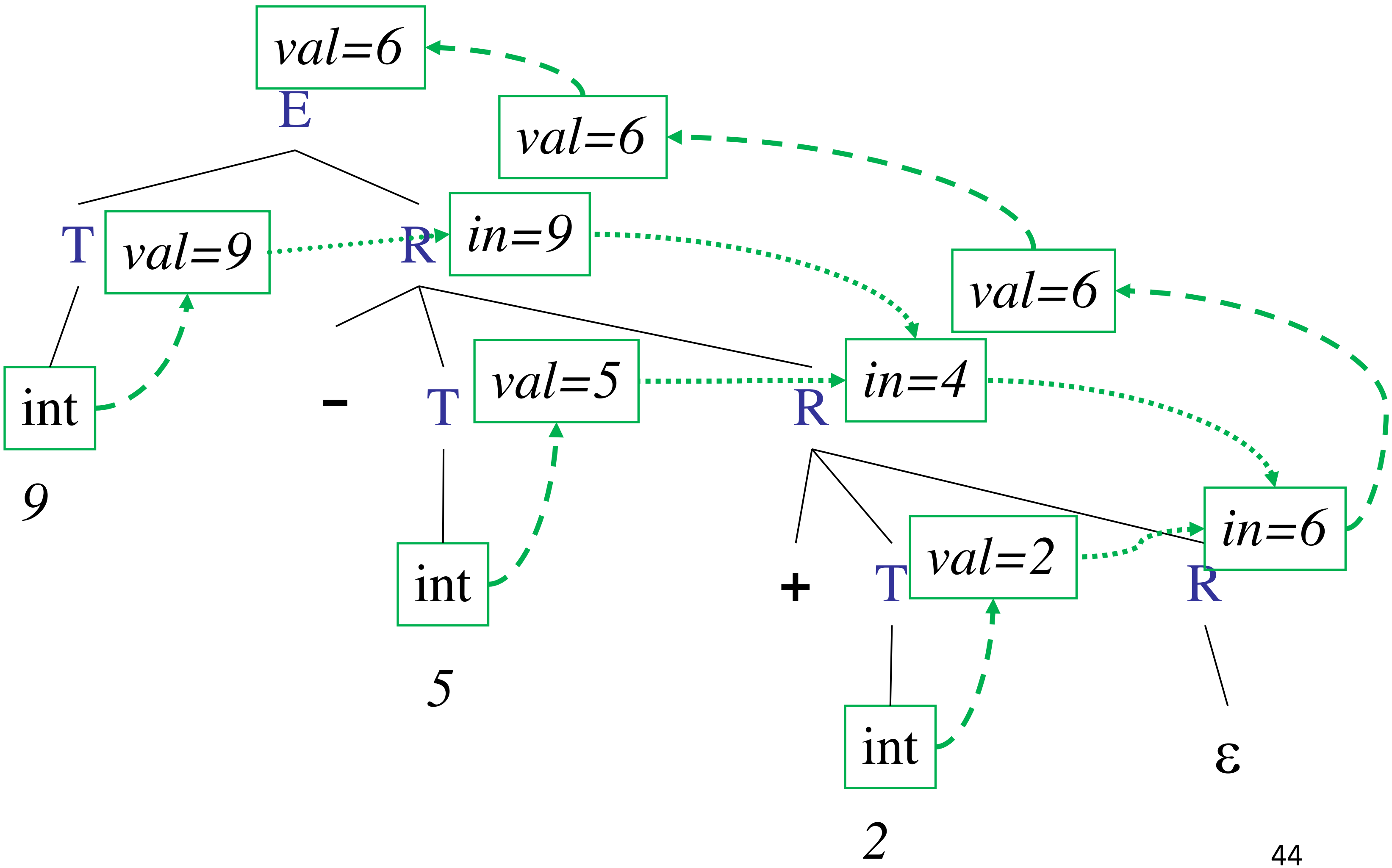
A $\rightarrow$ B { $0.val = $1.in; $1.in = $0.val + 1; }

- It is impossible to evaluate either $0.val or $1.in first (each value depends on the other)
- We want to avoid circular dependencies
- Detecting such cases in all parse trees takes exponential time!
- S-attributed or L-attributed definitions cannot have cycles

Dependency Graphs

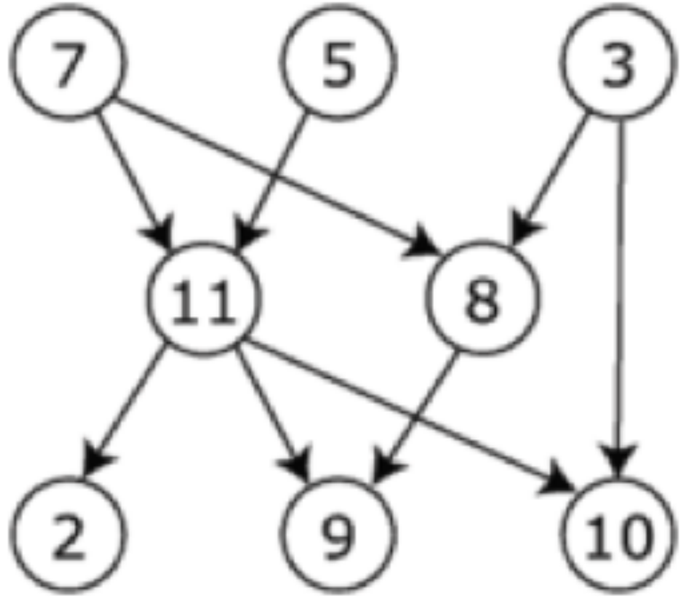
- Each dependency shows the flow of information in the parse tree
- All dependencies in each parse tree create a dependency graph

Dependency Graphs



Dependency Graphs

- Each dependency shows the flow of information in the parse tree
- All dependencies in each parse tree create a dependency graph
- Dependencies can be ordered and each ordering is called a **topological sort** of the dependency edges
- Each topological sort is a valid order of evaluation for semantic rules



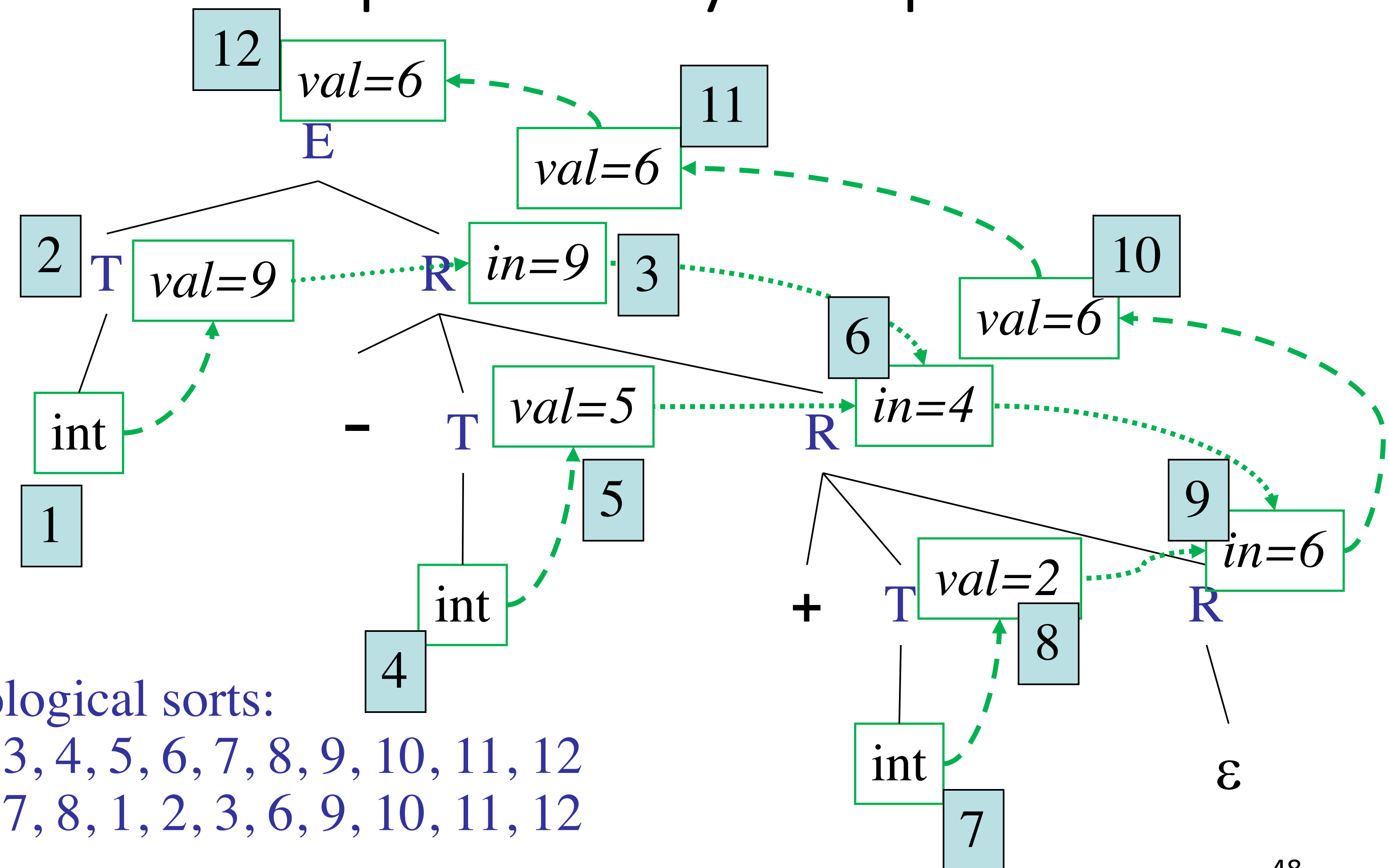
- A directed acyclic graph has many valid topological sort:
  - 7,5,3,11,8,2,9,10 (visual left-to-right top-to-bottom)
  - 3,5,7,8,11,2,9,10 (smallest-numbered available vertex first)
  - 3,7,8,5,11,10,2,9
  - 5,7,3,8,11,10,9,2 (least number of edges first)
  - 7,5,11,3,10,8,9,2 (largest-numbered available vertex first)
  - 7,5,11,2,3,8,9,10

Source: Wikipedia

Dependency Graphs

- Topological sort :
  - Order the nodes of the graph as $N_1, \ldots, N_k$ such that no edge in the graph goes from N_i to N_j ($j < i$)

Dependency Graphs



Topological sorts:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
4, 5, 7, 8, 1, 2, 3, 6, 9, 10, 11, 12

Syntax-directed definition with actions

- Some definitions can have side-effects:

$E \rightarrow T\ R$

{$2.in = $1.val; $0.val = $2.val; printf("%s", $2.in); }

- When will these side-effects occur?
- The order of evaluating attributes is linked to the order of creating nodes in the parse tree

Syntax-directed definition with actions

- A definition with side-effects:
  - $B \rightarrow X \{a\} Y$
- Bottom-up parser:
  - Perform action a as soon as X appears on top of the stack (if X is a nonterminal, we can move action a to $X \rightarrow A_1 \ldots A_2$ $\{a\}$)
- Top-down parser:
  - Perform action a just before we attempt to expand Y (if Y is a non-terminal) or check for Y on the input (if Y is a terminal)

SDTs with Actions

- A syntax directed definition with actions:

$E \rightarrow T\ R$

$R \rightarrow$ **+** T { print('+'); } R

$R \rightarrow$ **–** T { print('–'); } R

$R \rightarrow \varepsilon$

$T \rightarrow$ **int** { print(**int**.lookup); }

Stack	Input	Action
E $	9-5+2$	
T R $	9-5+2$	
int R $	9-5+2$	**print(9)**
R $	-5+2$	
- T R $	-5+2$	
T R $	5+2$	
int R $	5+2$	**print(5)**
R $	+2$	**print(-)**
+T R $	+2$	
T R$	2$	
int R$	2$	**print(2)**
R$	$	**print(+)**
$	$	**terminate**

Input:9 - 5 + 2

E → T R
R → + T { print('+'); } R
R → – T { print('–'); } R
R → ε
T → **int** { print(**int**.lookup); }

	+	-	int	$
E			**T R**	
T			**int**	
R	**+ T R**	**- T R**		**ε**

output: 9 5 - 2 +

SDT maps infix expressions to postfix

Actions in stack

- Action a for $B \rightarrow X \{a\} Y$ is pushed to the stack when the derivation step $B \rightarrow X \{a\} Y$ is made
- But the action is performed only after complete derivations for X has been carried out

Stack	Input	Action
E $	9-5+2$	
T R $	9-5+2$	
int R $	9-5+2$	**print(9)**
R $	-5+2$	
- T #A R $	-5+2$	
T #A R $	5+2$	
int #A R $	5+2$	**print(5)**
#A R $	+2$	**print(-)**
+T #B R $	+2$	
T #B R$	2$	
int #B R$	2$	**print(2)**
#B R$	$	**print(+)**
$	$	**terminate**

Input:9 - 5 + 2

$E \rightarrow T\ R$
$R \rightarrow +\ T$ { print('+'); } R
$R \rightarrow -\ T$ { print('–'); } R
$R \rightarrow \varepsilon$
$T \rightarrow$ **int** { print(**int**.lookup); }

	+	-	int	$
E			**T R**	
T			**int**	
R	**+ T R**	**- T R**		**ε**

#A: print('–')
#B: print('+')

SDTs with Actions

- Syntax directed definition that tries to map infix expressions to prefix:

E → T R

R → { print(‘+’); } + T R

R → { print(‘–’); } – T R

R → ε

T → **int** { print(**int**.lookup); }

Impossible to implement SDT during either top-down or bottom-up parsing, because the parser would have to perform printing actions long before it knows whether these symbols will appear in its input.

Marker non-terminals

- Bottom-up translation for L-attributed definitions:
- Assumption: each symbol X has one synthesized (X_{val}) and one inherited (X_{in}) attribute (or actions)

1. Replace each $A \rightarrow X_1 \ldots X_n$ by:

   $A \rightarrow M_1\, X_1 \ldots M_n\, X_n$, $M_i \rightarrow \varepsilon$ (new marker non-terminals)

2. When reducing by $M_i \rightarrow \varepsilon$

Compute $X_i.in$ ($M_i.val = X_i.in$)
push it into stack

M_i	$X_i.in$
X_{i-1}	$X_{i-1}.val$
M_{i-1}	$X_{i-1}.in$
⋮	⋮
X_1	$X_1.val$
M_1	$X_1.in$
M_A	$A.in$
⋮	⋮

Marker non-terminals

3. When reducing by $A \rightarrow M_1\ X_1 \ldots M_n\ X_n$:

   $A.val = f(M_1.val,\ X_1.val \ldots M_n.val,\ X_n.val)$

   Push A into stack

   $(M_i.val = X_i.in)$

X_n	$X_n.val$
M_n	$X_n\ .in$
:	:
X_1	$X_1.val$
M_1	$X_1.in$
M_A	$A.in$
:	:

A	$A.val$
M_A	$A.in$
:	:

4. Simplification: if X_j has no attributes or is computed by a copy rule $X_j.in = X_{j-1}.val$ discard M_j; adjust indices suitably. If $X_1.in$ exist and $X_1.in = A.in$, omit M_1

Marker Non-terminals

E → T R

R → + T { print(‘+’); } R

R → - T { print(‘-’); } R

R → ε

T → **int** { print(**int**.lookup); }

Marker Non-terminals

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R$

$R \rightarrow -\ T\ N\ R$

$R \rightarrow \varepsilon$

T → **int** { print(**int**.lookup); }

M → ε { print('+'); }

N → ε { print('-'); }

Equivalent SDT using *marker non-terminals*

Impossible Syntax-directed Definition

E → { print('+'); } E **+** T
E → { print('-'); } E **–** T
E → T
T → **(** E **)**
T → **int**{ print(**int**.lookup); }

Tries to convert infix to prefix

E → M E **+** T
E → N E **–** T
E → T
M→ ε { print('+'); }
N→ ε { print('-'); }
T → **(** E **)**
T → **int**{ print(**int**.lookup); }

Causes a reduce/reduce conflict when marker non-terminals are introduced.

Summary

- The parser produces concrete syntax trees
- Abstract syntax trees: define semantic checks or a syntax-directed translation to the desired output
- Attribute grammars: static definition of syntax-directed translation
  - Synthesized and Inherited attributes
  - S-attribute grammars
  - L-attributed grammars
- Complex inherited attributes can be defined if the full parse tree is available

Extra Slides

Syntax-directed defns

- LR parser, S-attributed definition
  - more details later …
- LL parser, L-attributed definition

Stack	Input	Output
$T')T'F	id)*id$	**T → F T' { $2.in = $1.val }**
$T')T'id	id)*id$	**F → id { $0.val = $1.val }**
$T')T'	)*id$	The action record stays on the stack when T' is replaced with rhs of rule

action record:
T'.in = F.val

LR parsing and inherited attributes

- As we just saw, inherited attributes are possible when doing top-down parsing
- How can we compute inherited attributes in a bottom-up shift-reduce parser
- Problem: doing it incrementally (while parsing)
- Note that LR parsing implies depth-first visit which matches L-attributed definitions

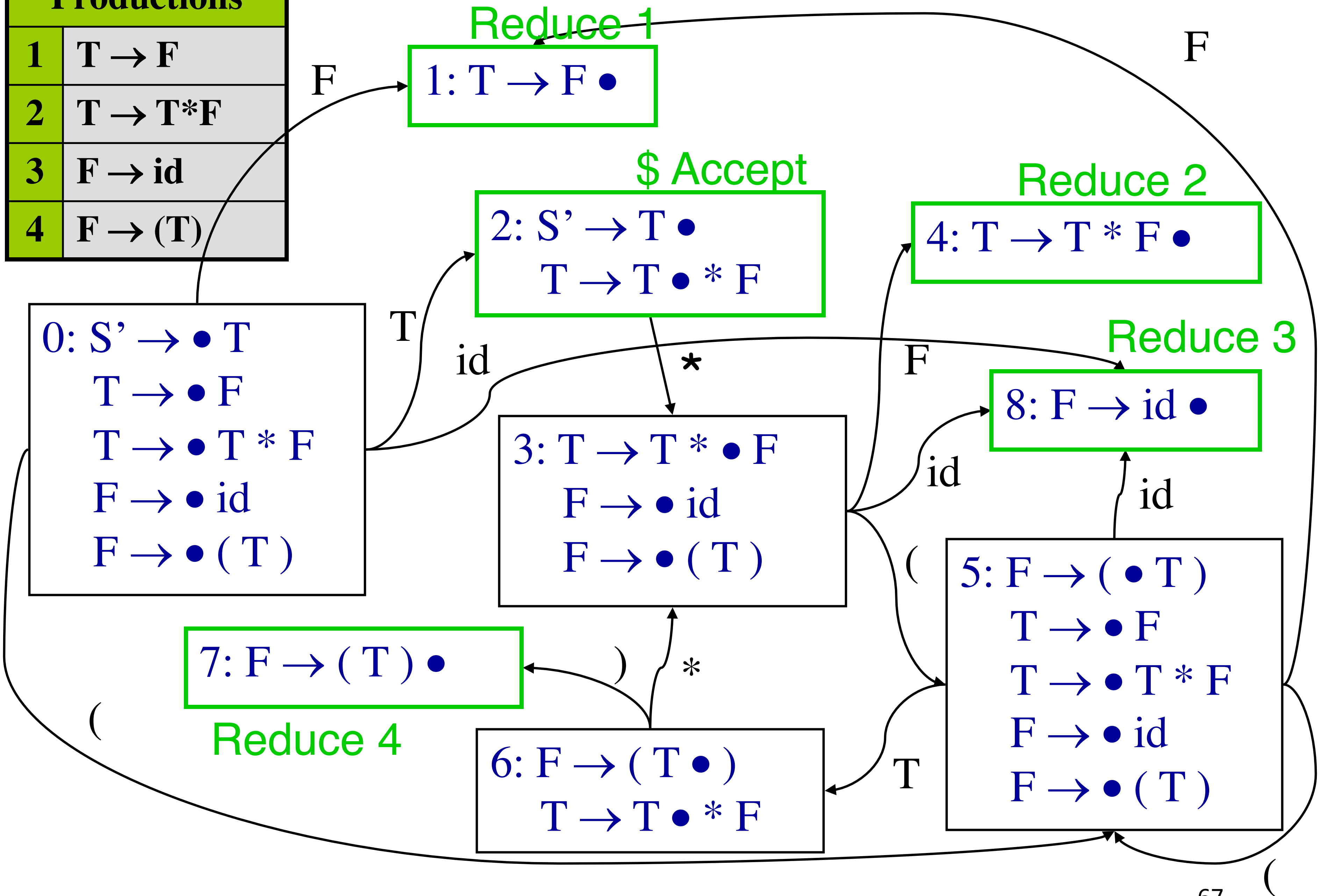
LR parsing and inherited attributes

- Attributes can be stored on the stack used by the shift-reduce parsing
- For synthesized attributes: when a reduce action is invoked, store the value on the stack based on value popped from stack
- For inherited attributes: transmit the attribute value when executing the **goto** function

Example: Synthesized Attributes

```
T → F    { $0.val = $1.val; }
T → T * F
  { $0.val = $1.val * $3.val; }
F → id
  { val := id.lookup();
    if (val) { $0.val = $1.val; }
    else { error; }
  }
F → ( T )   { $0.val = $2.val; }
```

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)



Trace "(id$_{val=3}$)*id$_{val=2}$"

Stack	Input	Action	Attributes
0	**(id) * id $**	**Shift 5**	
0 5	**id) * id $**	**Shift 8**	**a.Push id.val=3;**
0 5 8	**) * id $**	**Reduce 3 F→id,** **pop 8, goto [5,F]=1**	{ $0.val = $1.val } **a.Pop; a.Push 3;**
0 5 1	**) * id $**	**Reduce 1 T→ F,** **pop 1, goto [5,T]=6**	{ $0.val = $1.val }
0 5 6	**) * id $**	**Shift 7**	**a.Pop; a.Push 3;**
0 5 6 7	*** id $**	**Reduce 4 F→ (T),** **pop 7 6 5, goto [0,F]=1**	{ $0.val = $2.val } **3 pops; a.Push 3**

Trace “(id$_{val=3}$)*id$_{val=2}$”

Stack	Input	Action	Attributes
0 1	*** id $**	**Reduce 1 T→F, pop 1, goto [0,T]=2**	{ $0.val = $1.val } **a.Pop; a.Push 3**
0 2	*** id $**	**Shift 3**	**a.Push mul**
0 2 3	**id $**	**Shift 8**	**a.Push id.val=2**
0 2 3 8	**$**	**Reduce 3 F→id, pop 8, goto [3,F]=4**	**a.Pop a.Push 2**
0 2 3 4	**$**	**Reduce 2 T→T * F pop 4 3 2, goto [0,T]=2**	{ $0.val = $1.val * $3.val; }
0 2	**$**	**Accept**	**3 pops; a.Push 3*2=6**

Example: Inherited Attributes

$E \rightarrow$ T R

{ \$2.in = \$1.val; \$0.val = \$2.val; }

$R \rightarrow$ + T R

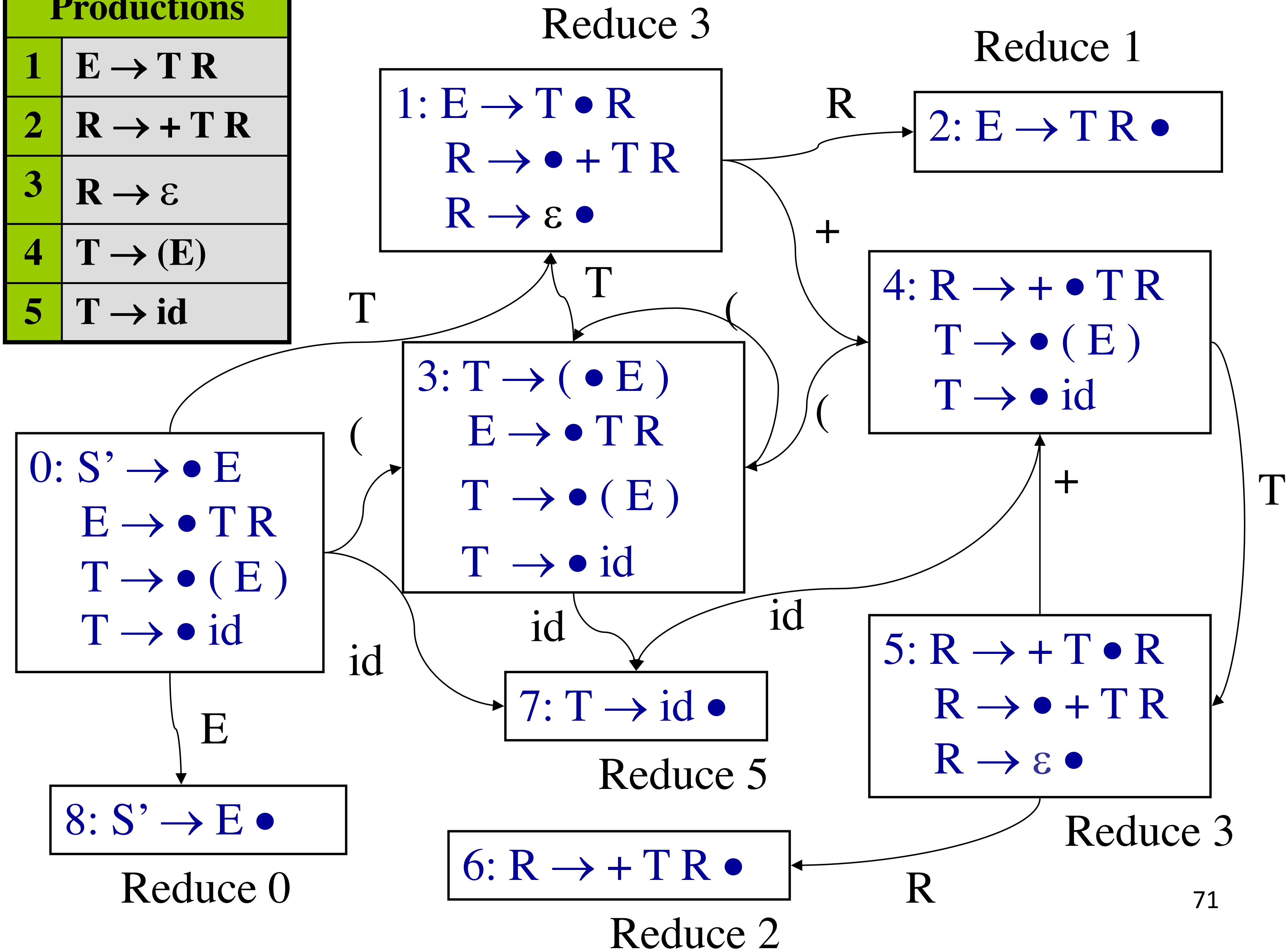
{ \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; }

$R \rightarrow \varepsilon$ { \$0.val = \$0.in; }

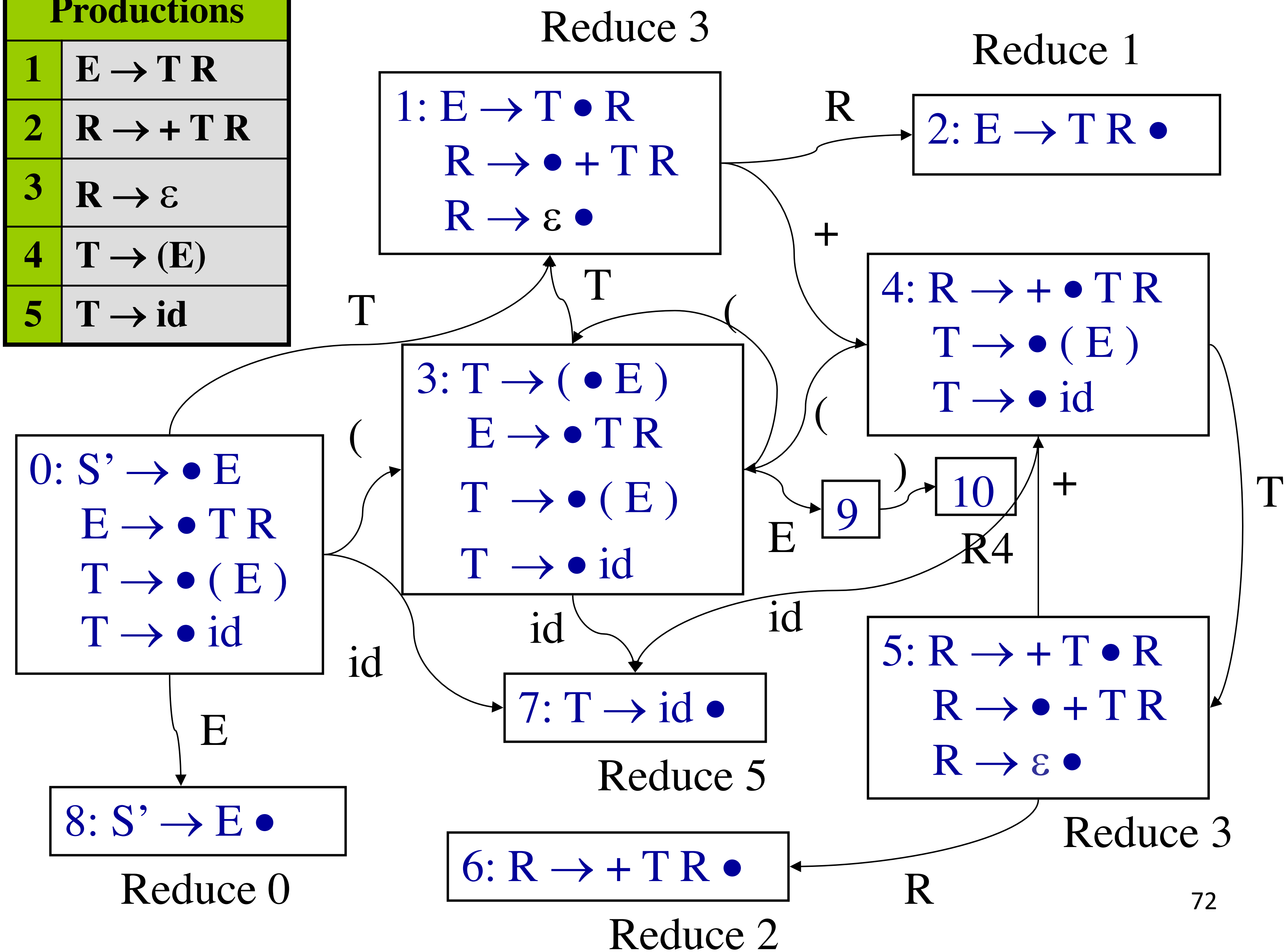
$T \rightarrow$ (E) { \$0.val = \$1.val; }

$T \rightarrow$ **id** { \$0.val = **id**.lookup; }

Productions	
1	E → T R
2	R → + T R
3	R → ε
4	T → (E)
5	T → id



Productions	
1	E → T R
2	R → + T R
3	R → ε
4	T → (E)
5	T → id



Productions	
1	**E → T R** { \$2.in = \$1.val; \$0.val = \$2.val; }
2	**R → + T R** { \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; }
3	**R →** ε { \$0.val = \$0.in; }
4	**T → (E)** { \$0.val = \$1.val; }
5	**T → id** { \$0.val = **id**.lookup; }

			ttributes
0 7	**+ id \$**	**Reduce 5 T→id**	**{ \$0.val = id.lookup }**
		pop 7, goto [0,T]=1	**{ pop; attr.Push(3)** **\$2.in = \$1.val** **\$2.in := (1).attr }**
0 1	**+ id \$**	**Shift 4**	
0 1 4	**id \$**	**Shift 7**	
0 1 4 7	**\$**	**Reduce 5 T→id**	**{ \$0.val = id.lookup }**
		pop 7, goto [4,T]=5	**{ pop; attr.Push(2); }**
0 1 4 5	**\$**	**Reduce 3 R→ ε** **goto [5,R]=6**	**{ \$3.in = \$0.in+\$1.val** **(5).attr := (1).attr+2** **\$0.val = \$0.in** **\$0.val = (5).attr7± 5 }**

Trace "$\text{id}_{val=3}$+$\text{id}_{val=2}$"

Stack	Input	Action	Attributes
0	**id + id $**	**Shift 7**	
0 7	**+ id $**	**Reduce 5 T→id** **pop 7, goto [0,T]=1**	**{ $0.val = id.lookup }** **{ pop; attr.Push(3)** **$2.in = $1.val** **$2.in := (1).attr }**
0 1	**+ id $**	**Shift 4**	
0 1 4	**id $**	**Shift 7**	
0 1 4 7	**$**	**Reduce 5 T→id** **pop 7, goto [4,T]=5**	**{ $0.val = id.lookup }** **{ pop; attr.Push(2); }**
0 1 4 5	**$**	**Reduce 3 R→ ε** **goto [5,R]=6**	**{ $3.in = $0.in+$1.val** **(5).attr := (1).attr+2** **$0.val = $0.in** **$0.val = (5).attr = 5 }**

Trace "$id_{val=3}$+$id_{val=2}$"

Stack	Input	Action	Attributes
0 1 4 5 6	**$**	**Reduce 2 R→ + T R** **Pop 4 5 6, goto [1,R]=2**	**{ $0.val = $3.val** **pop; attr.Push(5); }**
0 1 2	**$**	**Reduce 1 E→ T R** **Pop 1 2, goto [0,E]=8**	**{ $0.val = $3.val** **pop; attr.Push(5); }**
0 8	**$**	**Accept**	**{ $0.val = 5** **attr.top = 5; }**

A -> c { \$0.val = \$0.in }

LR parsing with inherited attributes

Bottom-Up/rightmost	
ccbca ⇐ Acbca	A→c
⇐ AcbB	B→ca
⇐ AB	B→cbB
⇐ S	S→AB

line 3

Parse stack at line 3:
[‘x’] A [‘x’] c b B

\$1.in = ‘x’

\$2.in = \$1.val

Consider:

S→AB
{ \$1.in = ‘x’;
\$2.in = \$1.val }

B→cbB
{ \$0.val = \$0.in + ‘y’; }

Parse stack at line 4:
[‘x’] A B

[‘xy’]