# Syntax Directed Translation for LR Parsers

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

# Syntax directed Translation

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

## Attribute Grammars

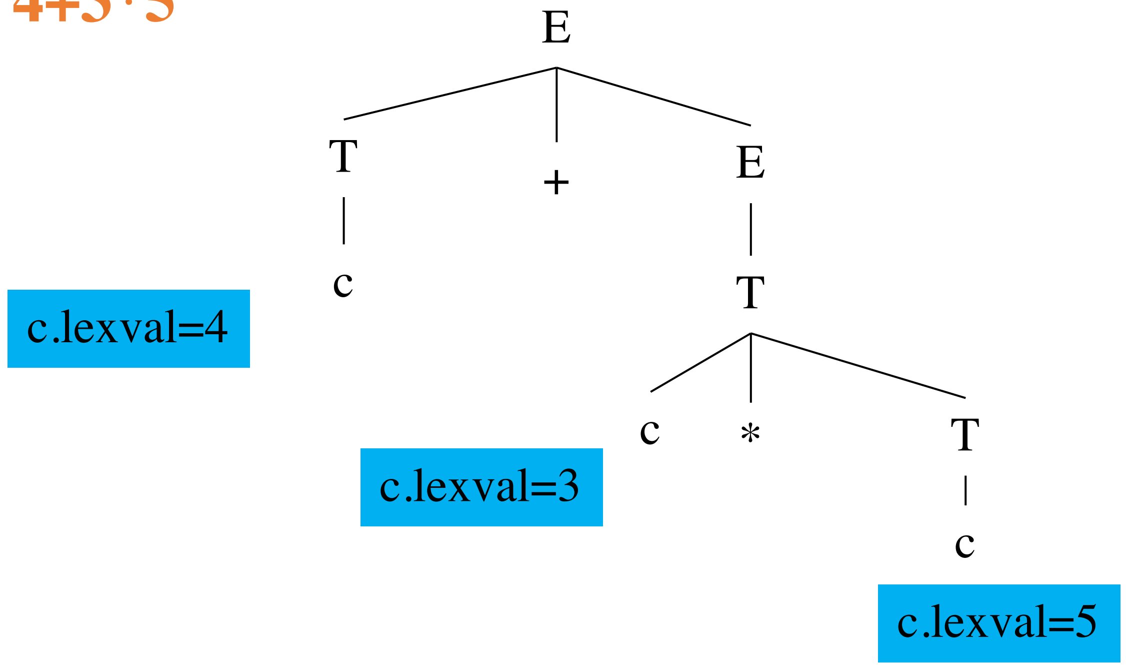
- Syntax-directed translation uses a grammar to produce code (or any other “semantics”)
- We are generalizing context-free grammars
- Each grammar symbol is associated with an attribute
- An attribute can be anything: a string, a number, a tree, any kind of record or object

# Attribute Grammars

- A CFG can be viewed as a function that relates strings to derivations (aka parse trees)
- Similarly, an attribute grammar is a way of relating strings with attributes (or “meanings”)
- Attribute grammars are a method to *decorate* or *annotate* the parse tree with the desired output attributes

## Expr concrete syntax tree

Input: **4+3*5**



$E \rightarrow T + E$

$E \rightarrow T$

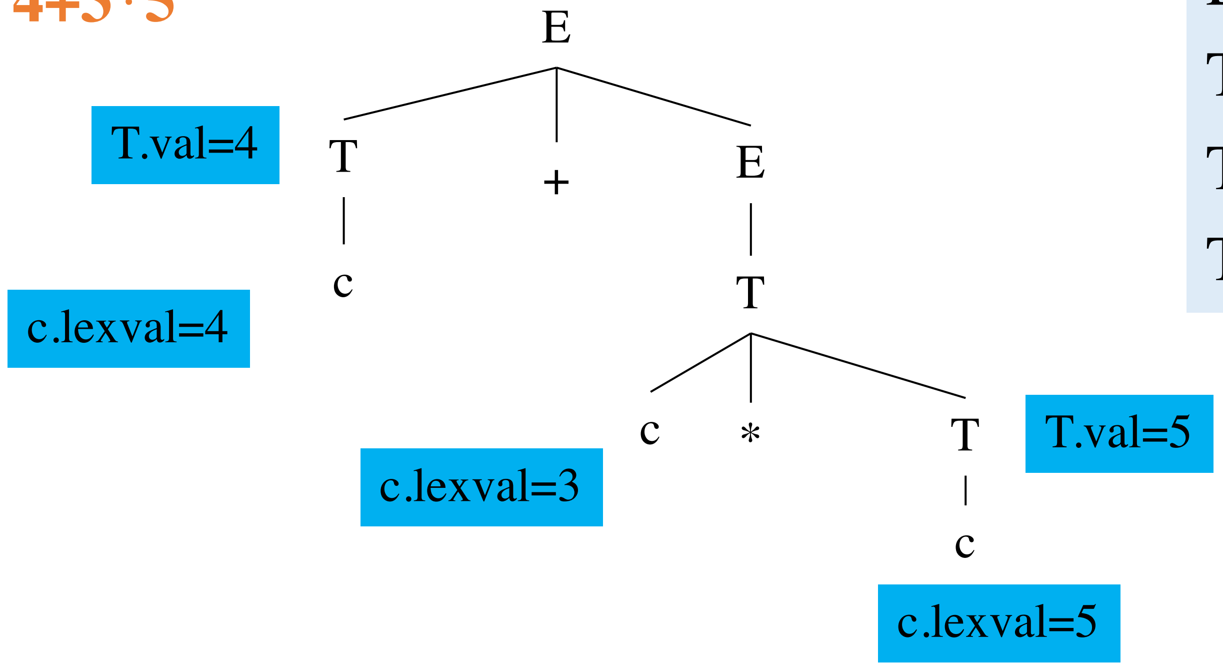
$T \rightarrow c$

$T \rightarrow c * T$

$T \rightarrow ( E )$

# Expr concrete syntax tree

Input: **4+3*5**



$E \rightarrow T + E$

$E \rightarrow T$

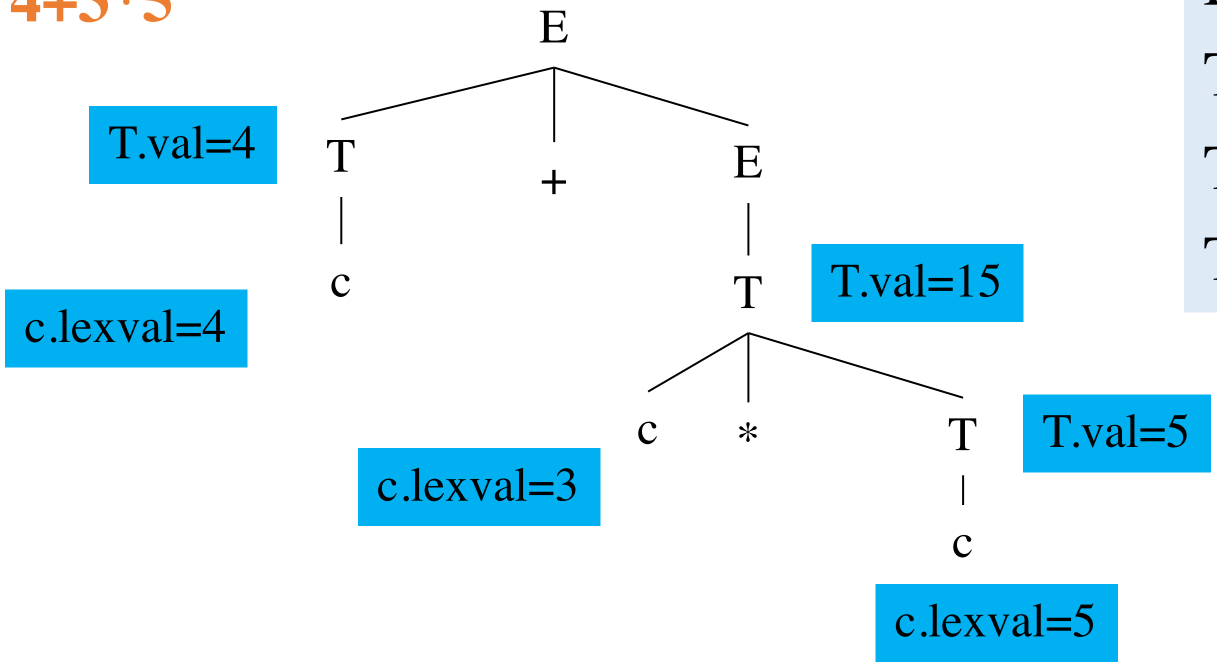
$T \rightarrow c$

$T \rightarrow c * T$

$T \rightarrow ( E )$

## Expr concrete syntax tree

Input: **4+3*5**



$E \rightarrow T + E$

$E \rightarrow T$

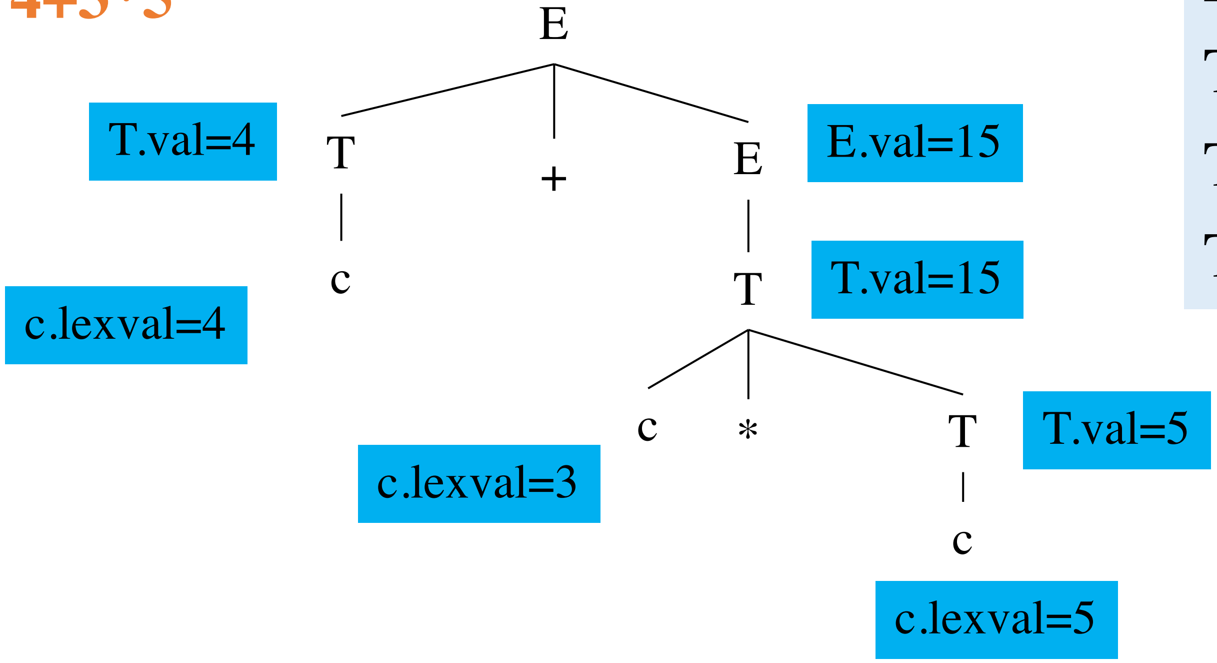
$T \rightarrow c$

$T \rightarrow c * T$

$T \rightarrow ( E )$

# Expr concrete syntax tree

Input: **4+3*5**



$E \rightarrow T + E$

$E \rightarrow T$

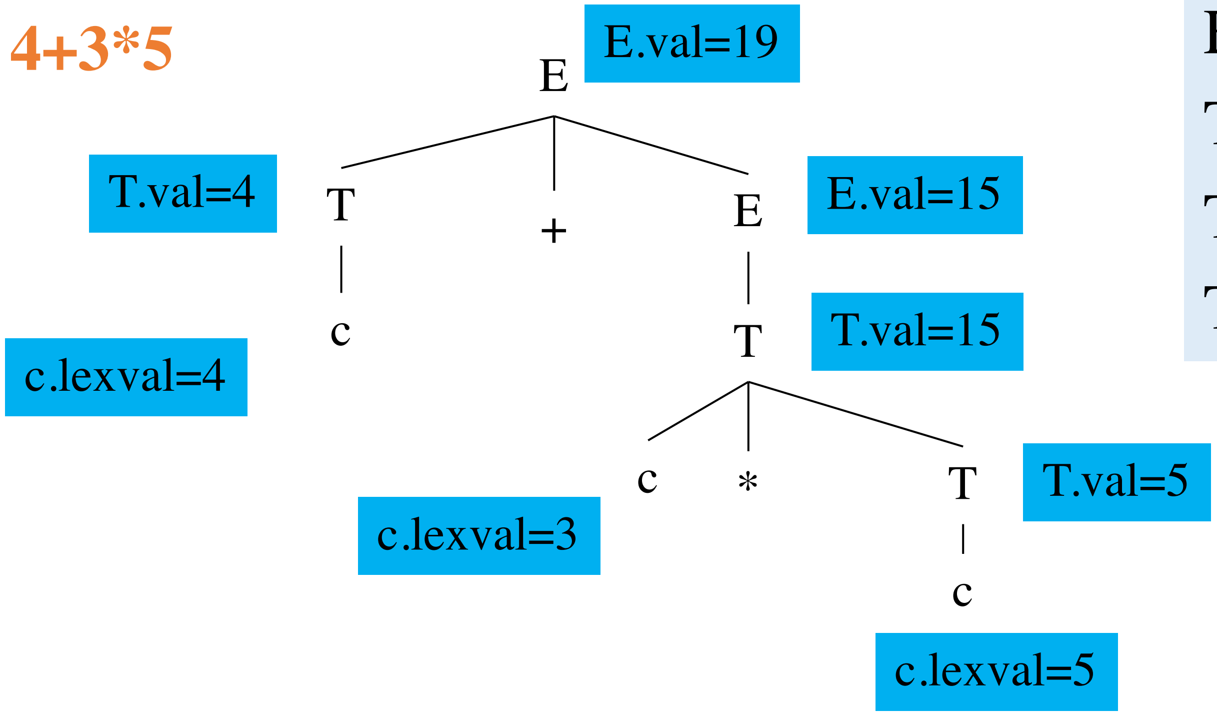
$T \rightarrow c$

$T \rightarrow c * T$

$T \rightarrow ( E )$

# Expr concrete syntax tree

Input: **4+3*5**



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow c$

$T \rightarrow c * T$

$T \rightarrow ( E )$

## Syntax directed definition

T → **c**

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

T → **c** ***** T

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

E → T

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

E → T **+** E

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

T → **(** E **)**

{ $$.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 IdList **;**

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

  IdList $\rightarrow$ **ID**

  IdList $\rightarrow$ **ID ,** IdList

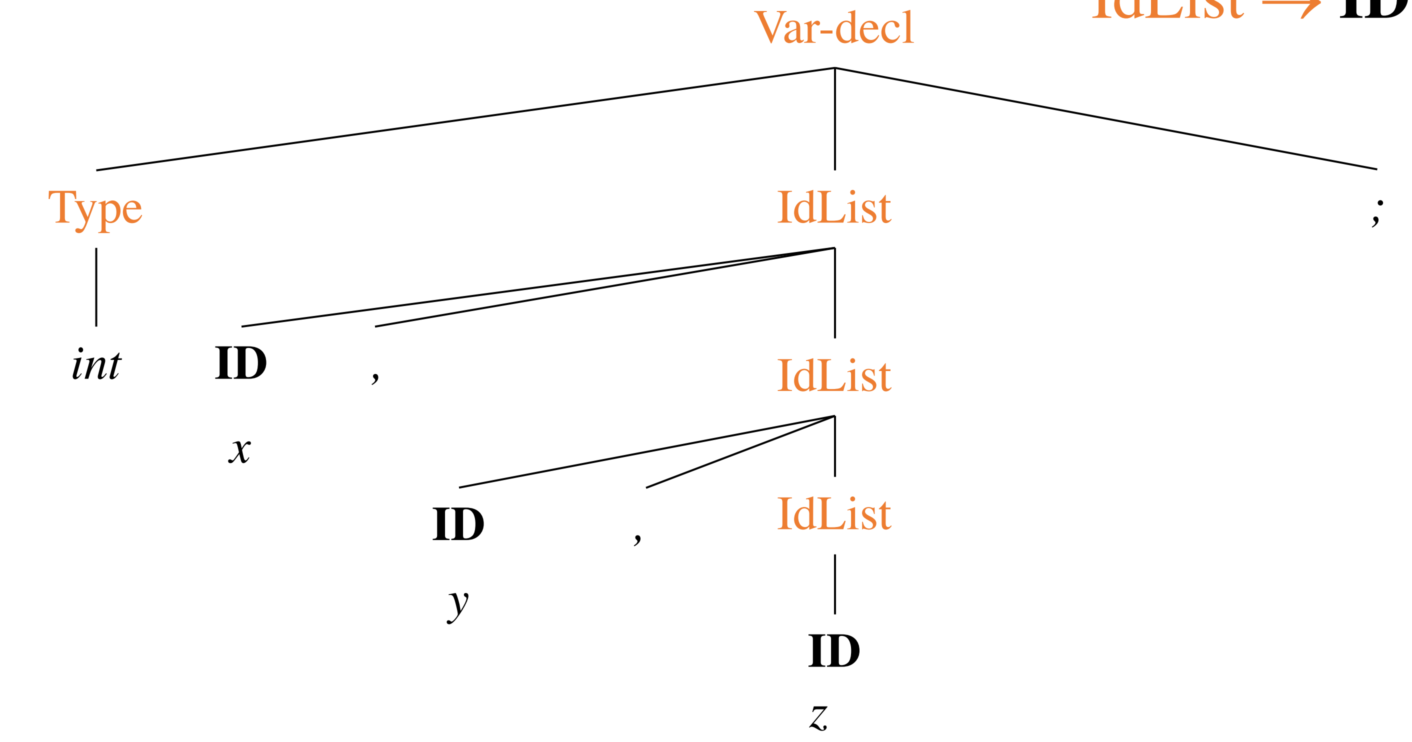
## Example input: *int x, y, z ;*

Var-decl → Type IdList

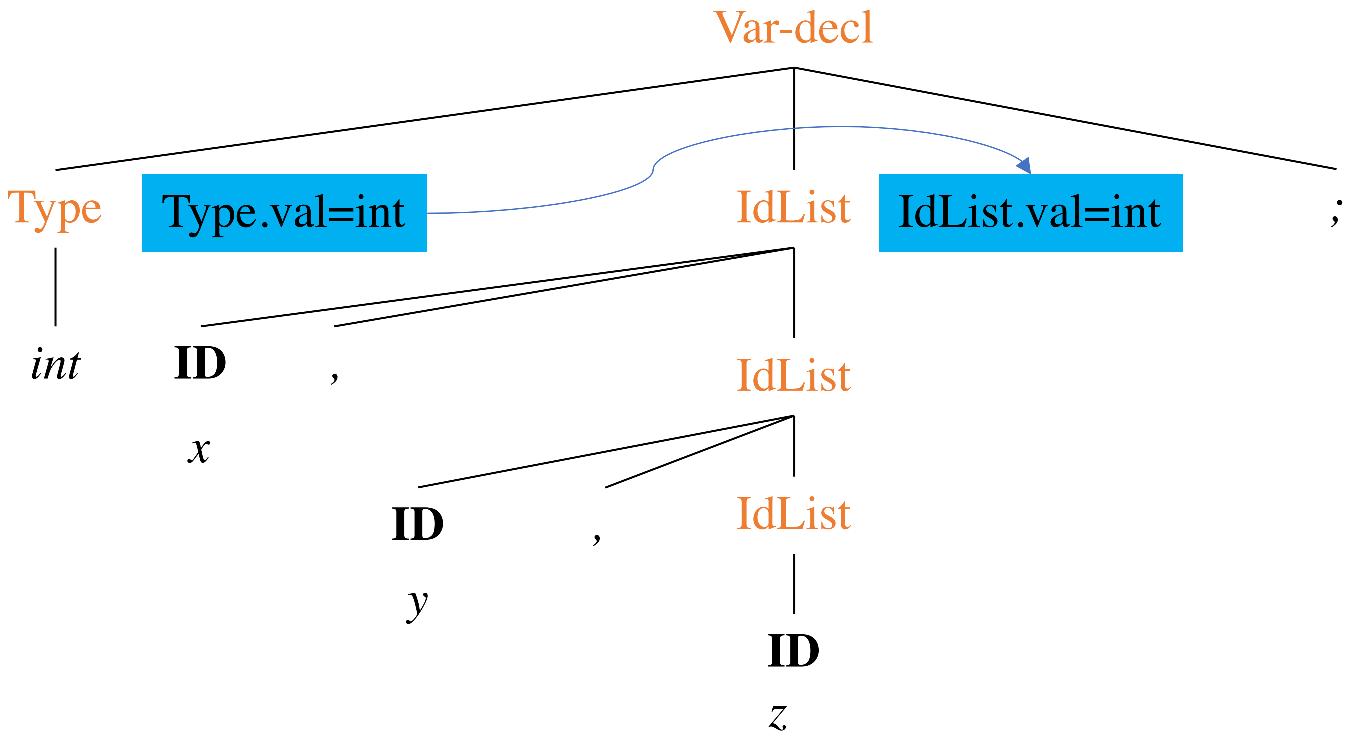
Type → ***int*** | ***bool***

IdList → **ID**

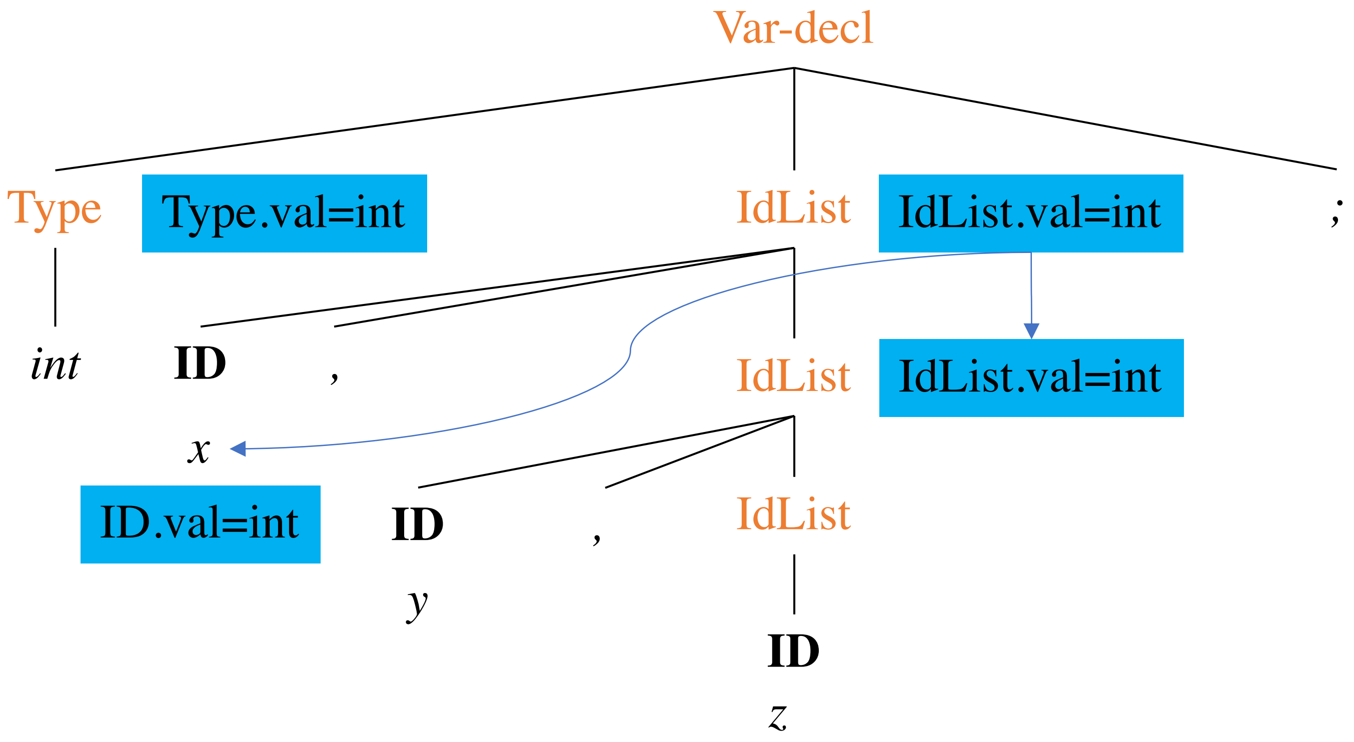
IdList → **ID , **IdList



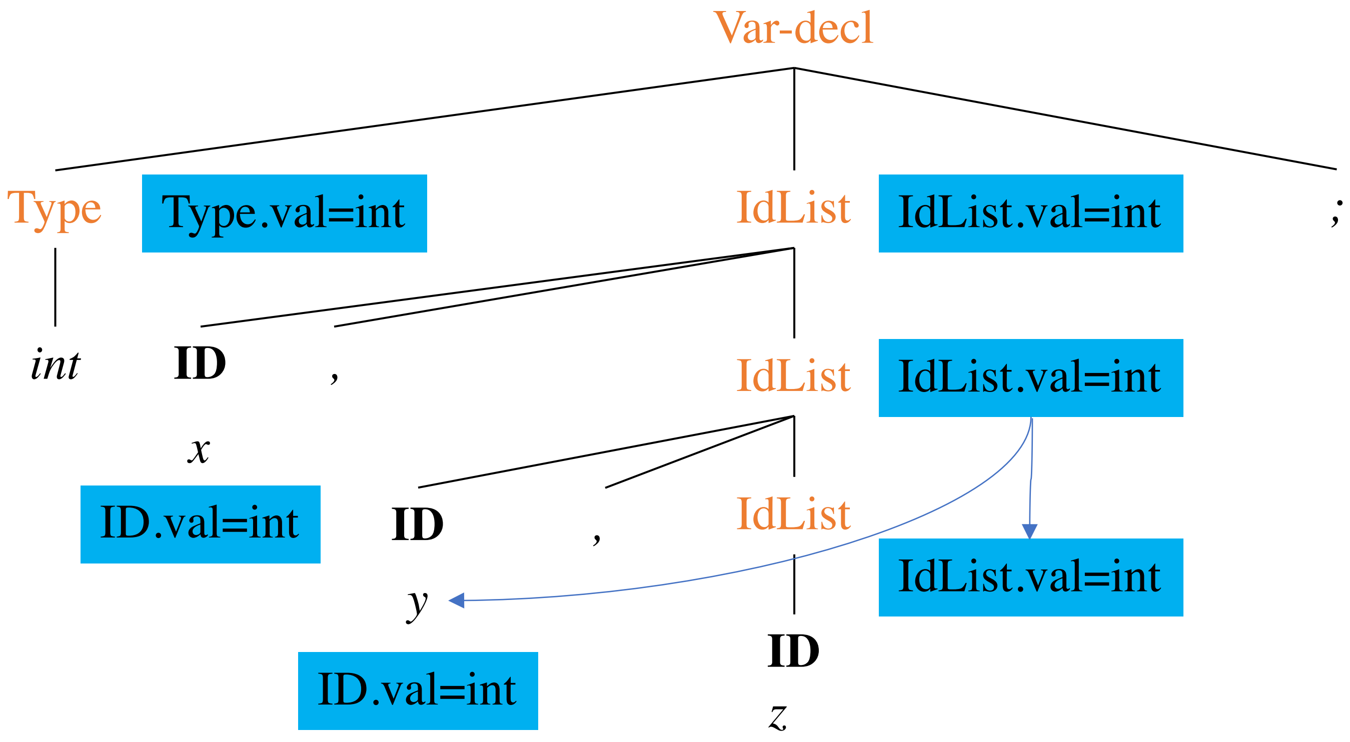
## Example input: *int x, y, z ;*



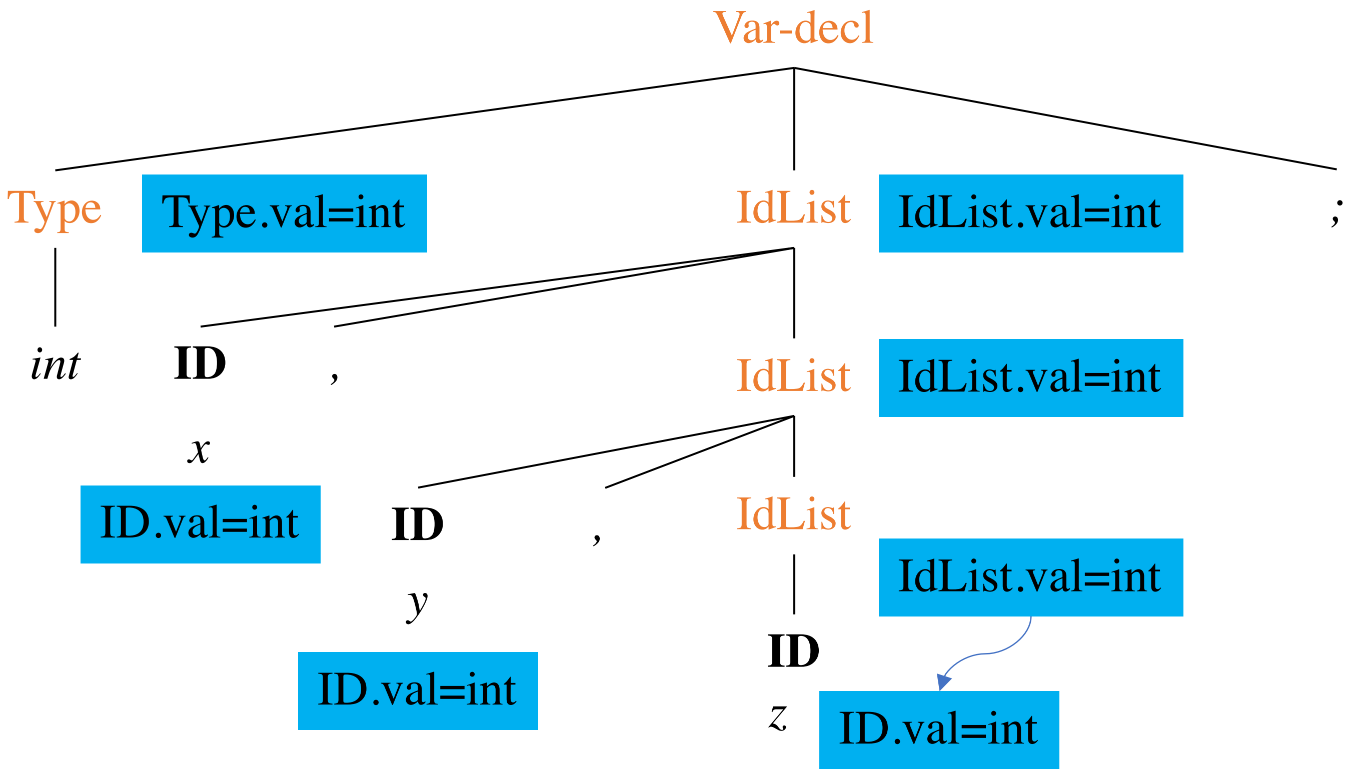
## Example input: *int x, y, z ;*



# 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
- *IdList* takes its attribute from its left sibling *Type*
- or *IdList* takes its attribute from its parent *IdList*

## Syntax-directed definition

Var-decl $\rightarrow$ Type IdList **;**

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

Type $\rightarrow$ **int**

{ $$.val = int; }

| **bool**

{ $$.val = bool; }

IdList $\rightarrow$ **ID**

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

IdList $\rightarrow$ **ID ,** IdList

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

Top-down (inheriting from the left-hand side) uses $0
Bottom-up (sending a value to the left-hand-side) uses $$

## Inherited Attributes

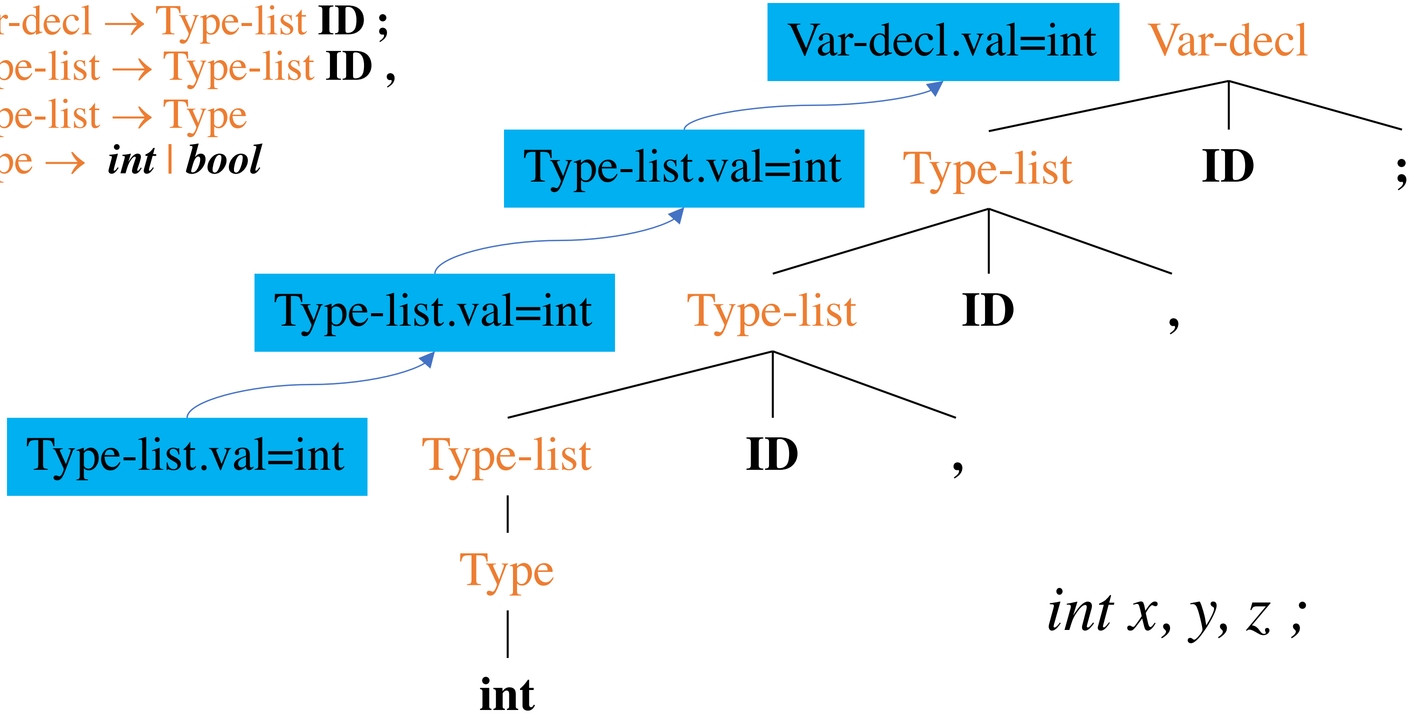
- **Inherited attributes** are attributes that are computed at a node based on attributes from siblings or the parent
- Typically we combine synthesized attributes and inherited attributes
- Q: It is possible to convert the grammar into a form that *only* uses synthesized attributes?

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

*int x, y, z ;*

# Removing Inherited Attributes

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



*int x, y, z ;*

## Removing inherited attributes

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

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

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

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

Type-list $\rightarrow$ Type

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

Type $\rightarrow$ **int**

{ $$.val = int; }

| **bool**

{ $$.val = bool; }

## Direction of inherited attributes

- Consider the syntax directed defns:

  $A \rightarrow L\ M$

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

  $A \rightarrow Q\ R$

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

- Problematic definition: \$1.in = \$2.val
- Incompatible with incremental processing (left to right parsing)

## 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

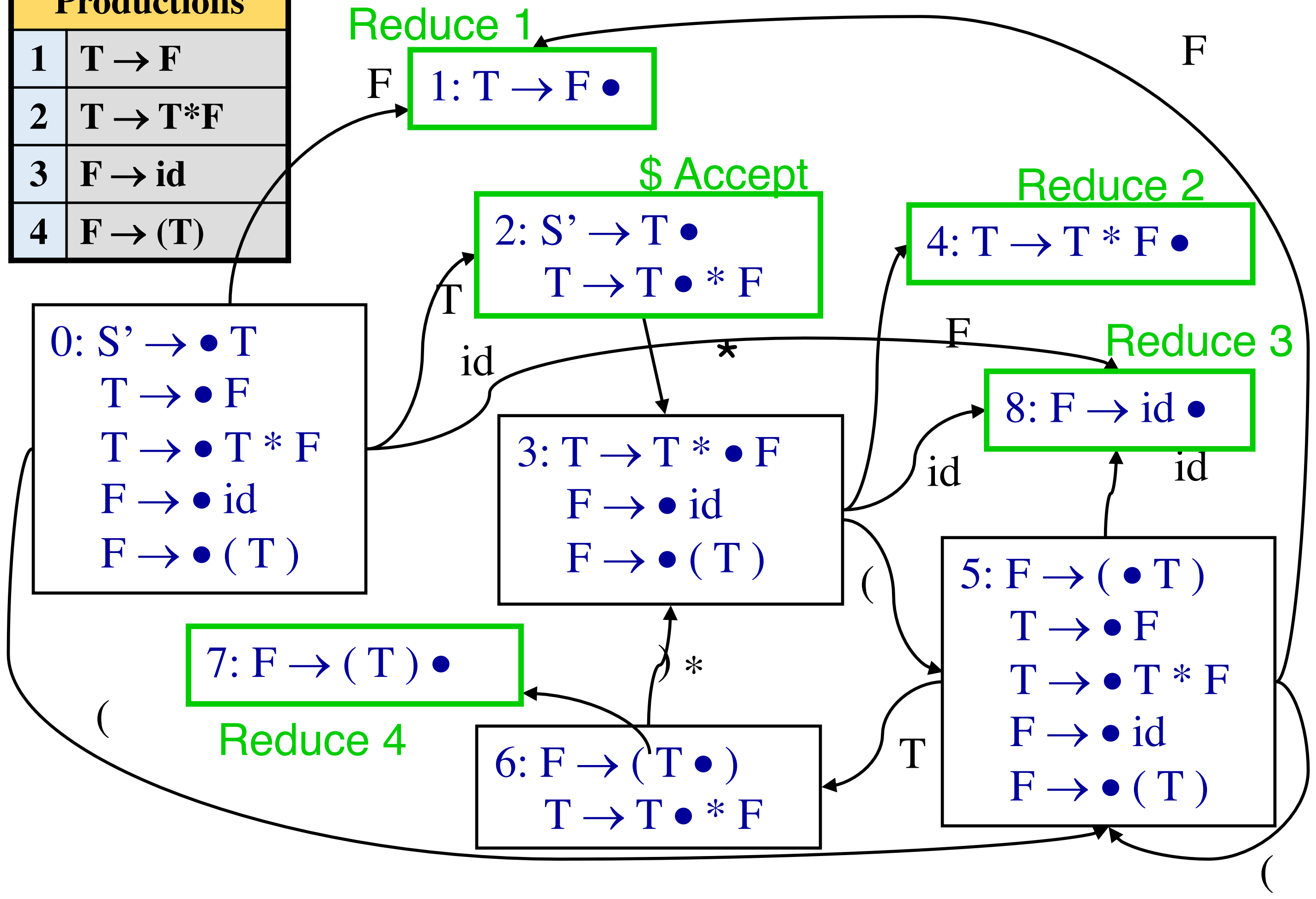
## LR parsing and attribute grammars

- LR parsing is inherently left to right
- Attributes can be stored on the stack used by 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    { $$.val = $1.val; }
T → T * F
  { $$.val = $1.val * $3.val; }
F → id
  { val := id.lookup();
     if (val) { $$.val = $1.val; }
     else { error; }
  }
F → ( T )  { $$.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 | Attribute Stack |
|---|---|---|---|
| **0** | **( id ) * id $** | **Shift 5** | |
| **0 5** | **id ) * id $** | **Shift 8** | **a.Push(id.val==3);** |
| **0 5 8** | **) * id $** | **Reduce 3 F→id,**<br>**pop 8, goto [5,F]=1** | { $$.val = $1.val }<br>**a.Push(a.Pop==3);** |
| **0 5 1** | **) * id $** | **Reduce 1 T→ F,**<br>**pop 1, goto [5,T]=6** | { $$.val = $1.val }<br>**a.Push(a.Pop==3);** |
| **0 5 6** | **) * id $** | **Shift 7** | |
| **0 5 6 7** | *** id $** | **Reduce 4 F→ (T),**<br>**pop 7 6 5, goto [0,F]=1** | { $$.val = $2.val }<br>**3 pops; a.Push(3)** |

## Trace “($id_{val=3}$)*$id_{val=2}$”

| Stack | Input | Action | Attribute Stack |
|---|---|---|---|
| 0 1 | * id $ | Reduce 1 T$\to$F, | { $$.val = $1.val } |
| | | pop 1, goto [0,T]=2 | a.Push(a.Pop==3) |
| 0 2 | * id $ | Shift 3 | a.Push(*) |
| 0 2 3 | id $ | Shift 8 | a.Push(id.val==2) |
| 0 2 3 8 | $ | Reduce 3 F$\to$id, | |
| | | pop 8, goto [3,F]=4 | a.Push(a.Pop==2) |
| 0 2 3 4 | $ | Reduce 2 T$\to$T * F | { $$.val = $1.val * $3.val; } |
| | | pop 4 3 2, goto [0,T]=2 | 3 pops; a.Push(3*2==6) |
| 0 2 | $ | Accept | return(6) |

## Practice question

$S \rightarrow L\,.\,L$
$S \rightarrow L$
$L \rightarrow L\,B$
$L \rightarrow B$
$B \rightarrow 0$
$B \rightarrow 1$

This grammar generates binary floating-point numbers, e.g. 101.101

Q: Write down an attribute grammar (syntax directed translation) that converts the input binary into decimal.

e.g. 101.101 = $5\frac{5}{8}$ = 5.625

integer part: $1\times2^2+0\times2^1+1\times2^0 = 5$

fractional part: $1\times\frac{1}{2^1}+0\times\frac{1}{2^2}+1\times\frac{1}{2^3} = \frac{5}{8}$

# Practice question

$$
\begin{array}{lcl}
E & \rightarrow & E \text{ '+' } T \\
  & | & T \\
T & \rightarrow & T \text{ '*' } F \\
  & | & F \\
F & \rightarrow & \texttt{exp} \text{ '(' } E \text{ ')' } \\
  & | & \texttt{ln} \text{ '(' } E \text{ ')' } \\
  & | & \text{'-' } F \\
  & | & \text{'x'} \\
  & | & \text{c}
\end{array}
$$

c stands for any integer constant

Provide a L-attributed syntax directed definition that computes the derivative of an input expression. Explain each attribute used in your attribute grammar.

| $D$[input string] | output string = derivative(input string) |
|---|---|
| $D[c]$ | $0$ |
| $D[x]$ | $1$ |
| $D[x+c]$ | $1$ |
| $D[E_1+E_2]$ | $D[E_1]+D[E_2]$ |
| $D[-E]$ | $-D[E]$ |
| $D[c*E]$ | $c*D[E]$ |
| $D[E_1*E_2]$ | $E_1*D[E_2]+E_2*D[E_1]$ |
| $D[exp(x)]$ | $exp(x)$ |
| $D[ln(x)]$ | $1/x$ |
| $D[f(E)]$ | $D[E]*f'(E)$, $f'$ is the derivative of $f$<br>if $f(E)$ is $exp(E)$, $f'(E)$ is $exp(E)$<br>if $f(E)$ is $ln(E)$, $f'(E)$ is $1/E$ |

# Extra Slides

## Example: Inherited Attributes

E $\rightarrow$ T R

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

R $\rightarrow$ + T R

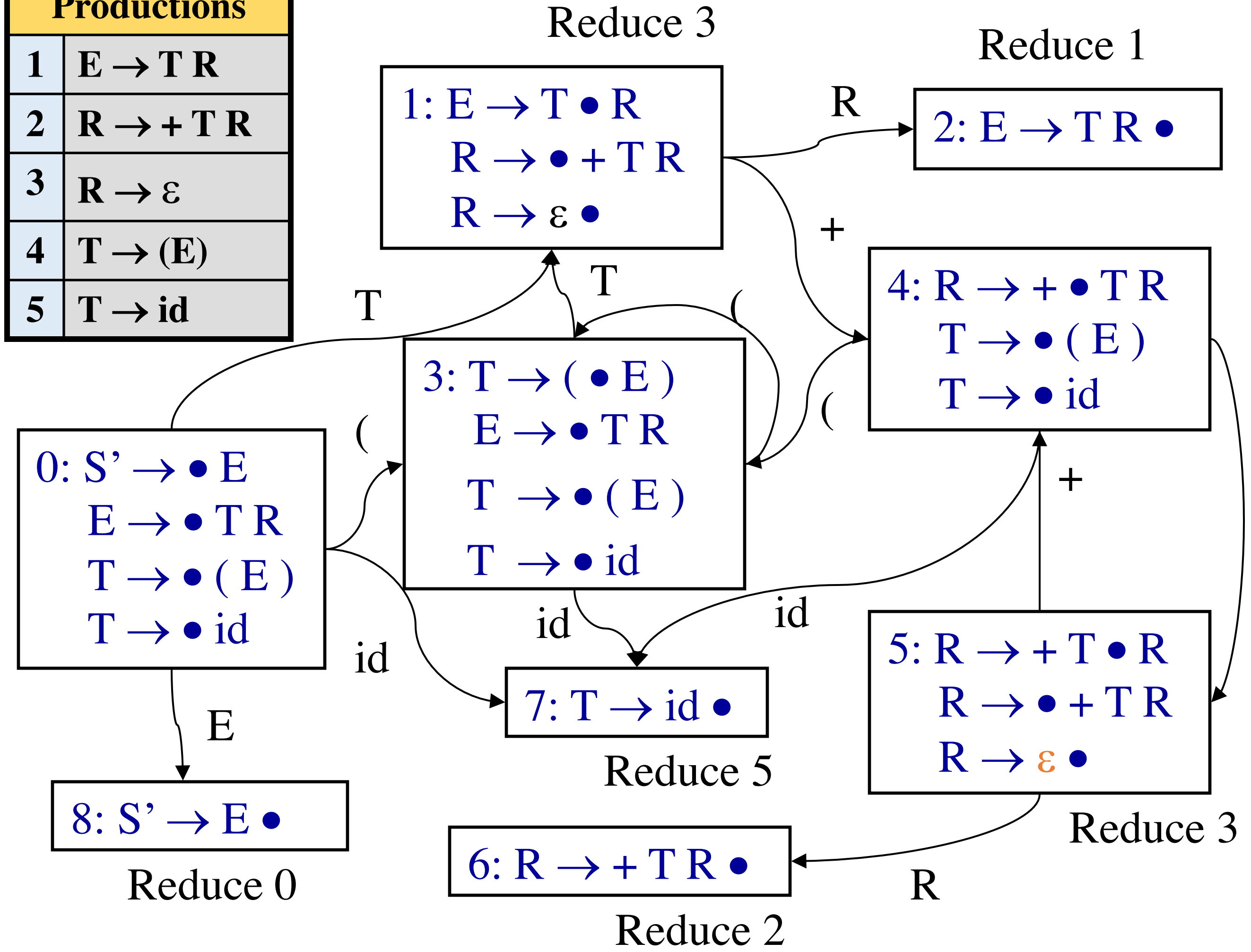
{ $3.in = $0.in + $2.val;  $$.val = $3.val; }

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

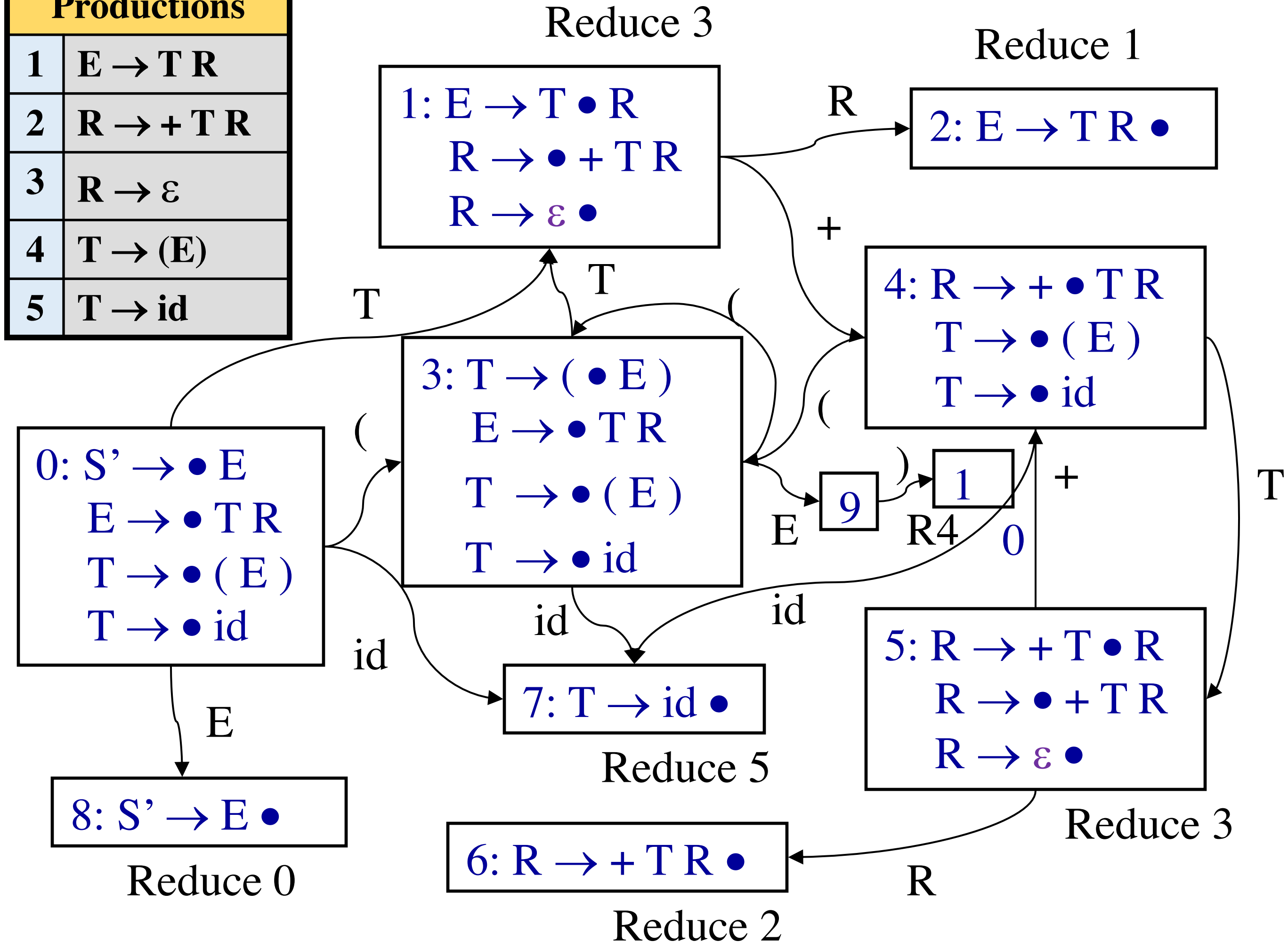
T $\rightarrow$ ( E )  { $$.val = $1.val; }

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

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

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

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

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

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

# LR parsing with inherited attributes

| Bottom-Up/rightmost | |
|---|---|
| ccbca $\Leftarrow$ Acbca | $A \to c$ |
| $\Leftarrow$ AcbB | $B \to ca$ |
| $\Leftarrow$ AB | $B \to cbB$ |
| $\Leftarrow$ S | $S \to AB$ |

line 3

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

$1.in = ‘x’

$2.in = $1.val

A → c { $$.val = $0.in }

Consider:

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

B→cbB
{ $$.val = $0.in + ‘y’; }

Parse stack at line 4:
[‘x’] A B

[‘xy’]