CMPT 379
Compilers

Anoop Sarkar

`http://www.cs.sfu.ca/~anoop`

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

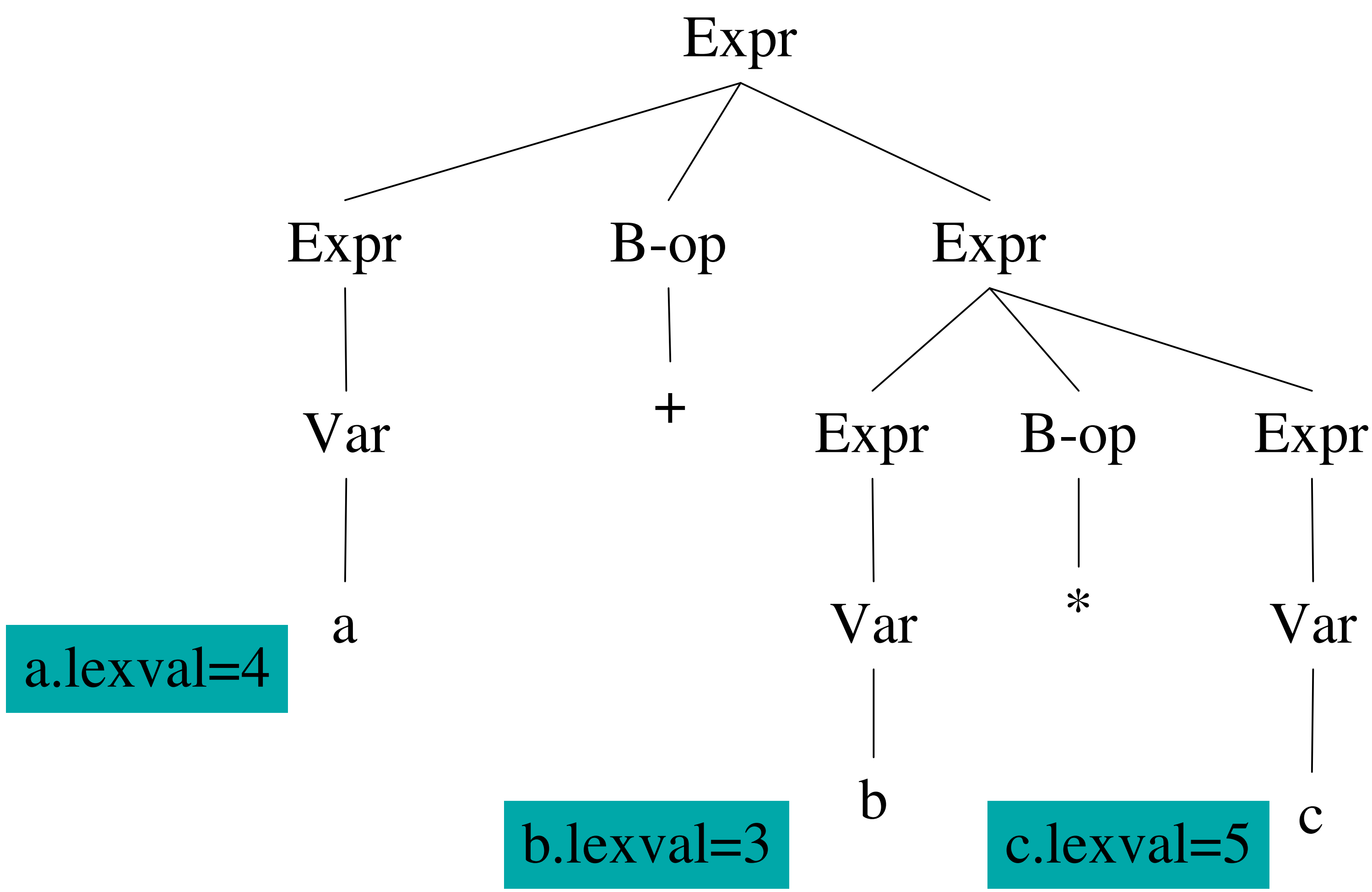
Attribute Grammars

- 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

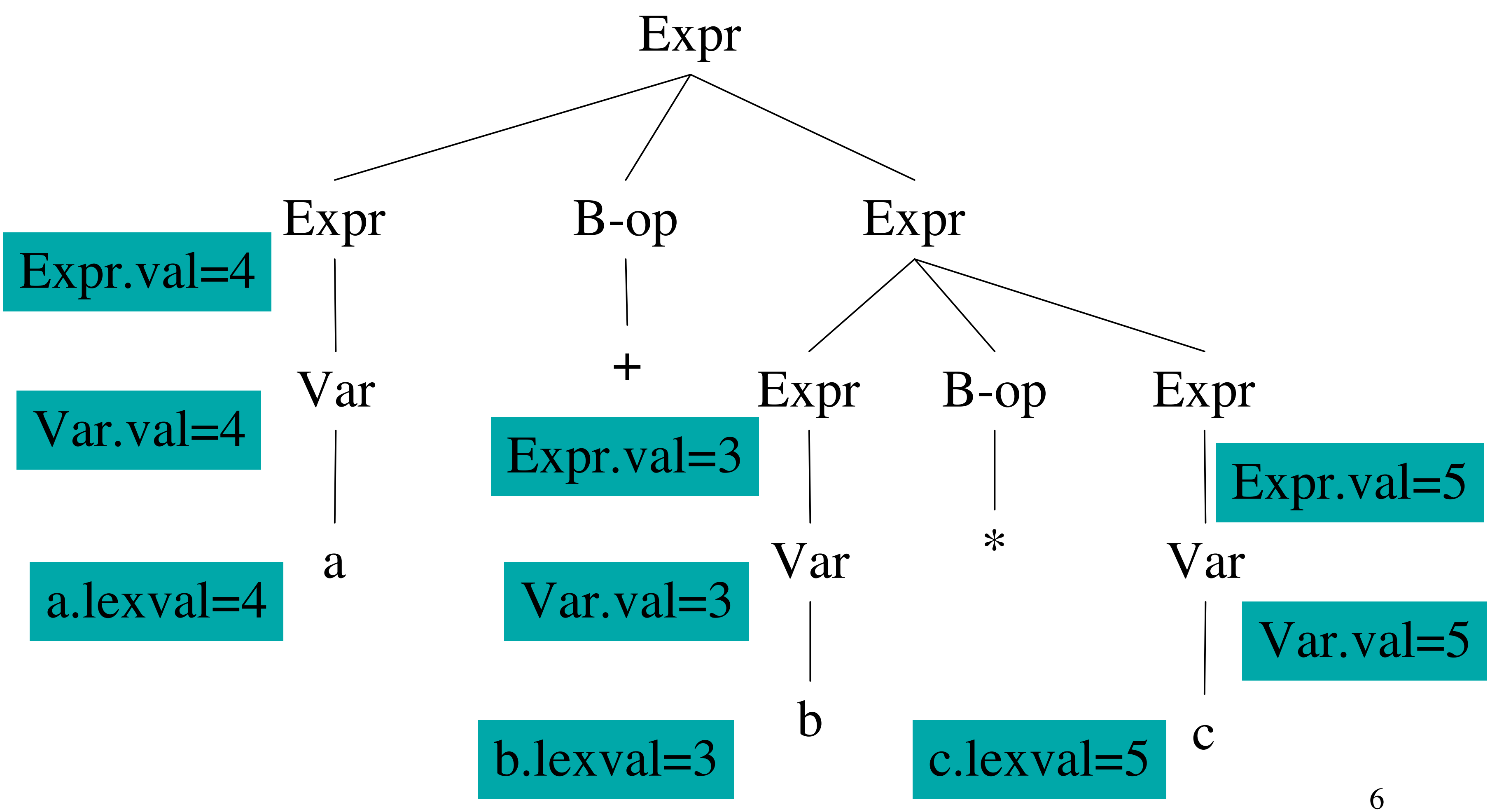
Attribute Grammars

- 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 semantics (rules to build an abstract syntax tree)
- In other words, attribute grammars are a method to *decorate* or *annotate* the parse tree

Example

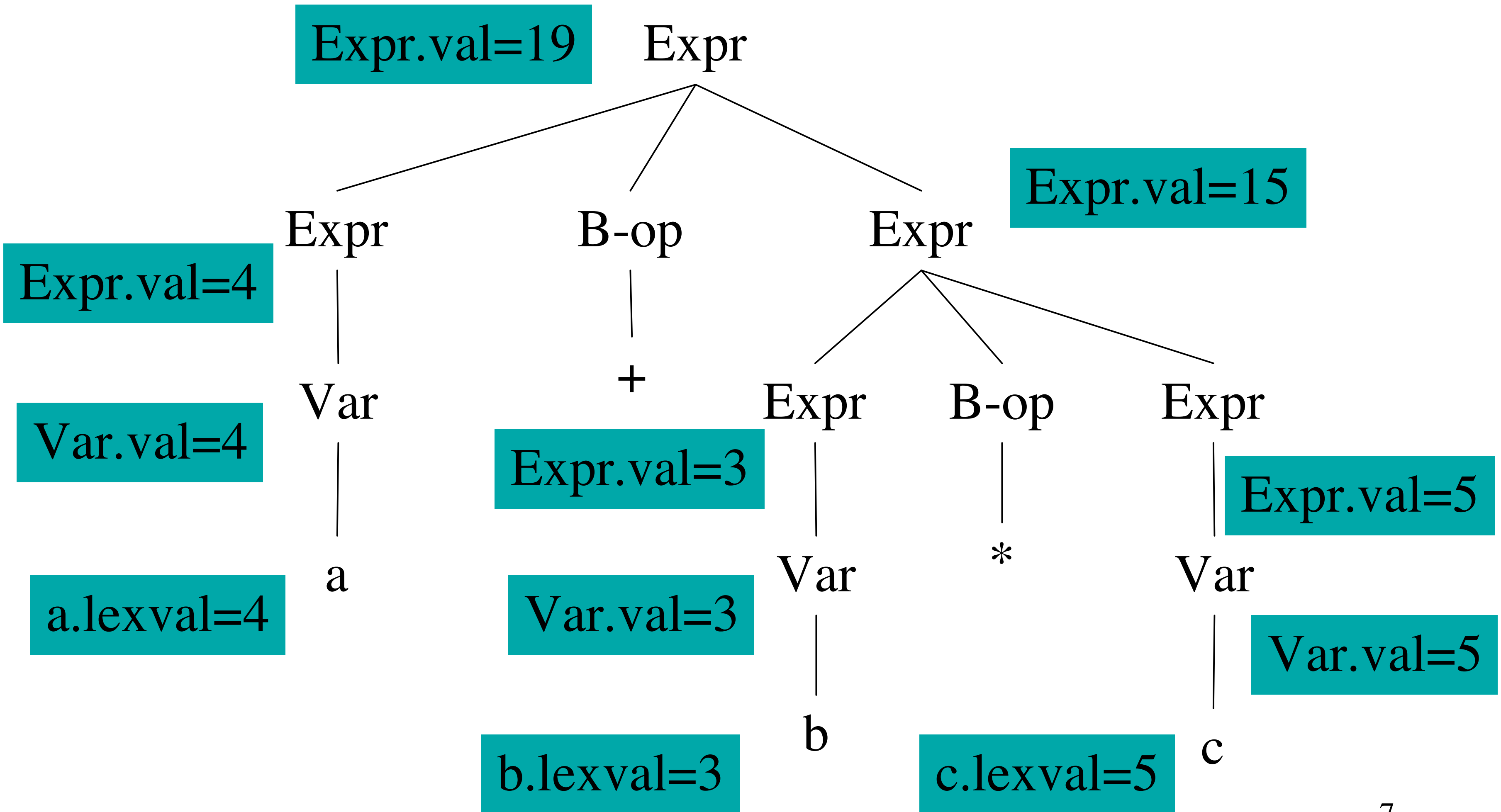

Expr
Expr
B-op
Expr
Var
+
Expr
B-op
Expr
a
Var
*
Var
a.lexval=4
b
c
b.lexval=3
c.lexval=5

Example



Expr
Expr
B-op
Expr
Expr.val=4
Var
+
Expr
B-op
Expr
Var.val=4
Expr.val=3
Expr.val=5
a
Var
*
Var
a.lexval=4
Var.val=3
Var.val=5
b
c
b.lexval=3
c.lexval=5

Example



Expr.val=19
Expr
Expr
B-op
Expr
Expr.val=15
Expr.val=4
Var
+
Expr
B-op
Expr
Var.val=4
Expr.val=3
Expr.val=5
a
Var
*
Var
a.lexval=4
Var.val=3
Var.val=5
b
c
b.lexval=3
c.lexval=5

Syntax directed definition

Var → IntConstant

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

Expr → Var

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

Expr → Expr B-op Expr

{ $0.val = $2.val ($1.val, $3.val); }

B-op → +

{ $0.val = PLUS; }

B-op → *

{ $0.val = TIMES; }

Flow of Attributes in *Expr*

- Consider the flow of the attributes in the *Expr* 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:

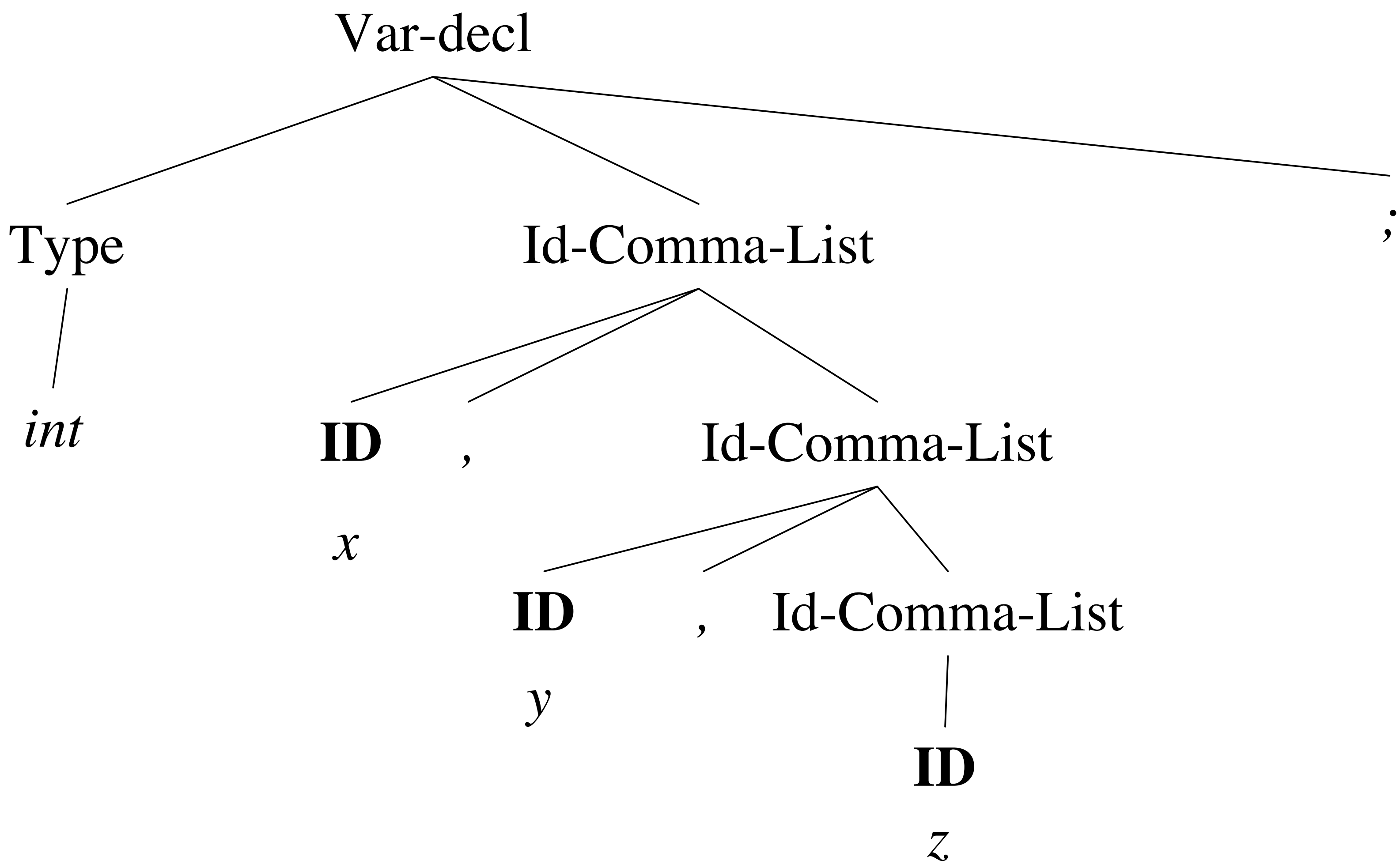
  Var-decl → Type Id-comma-list **;**

  Type → **int** | **bool**

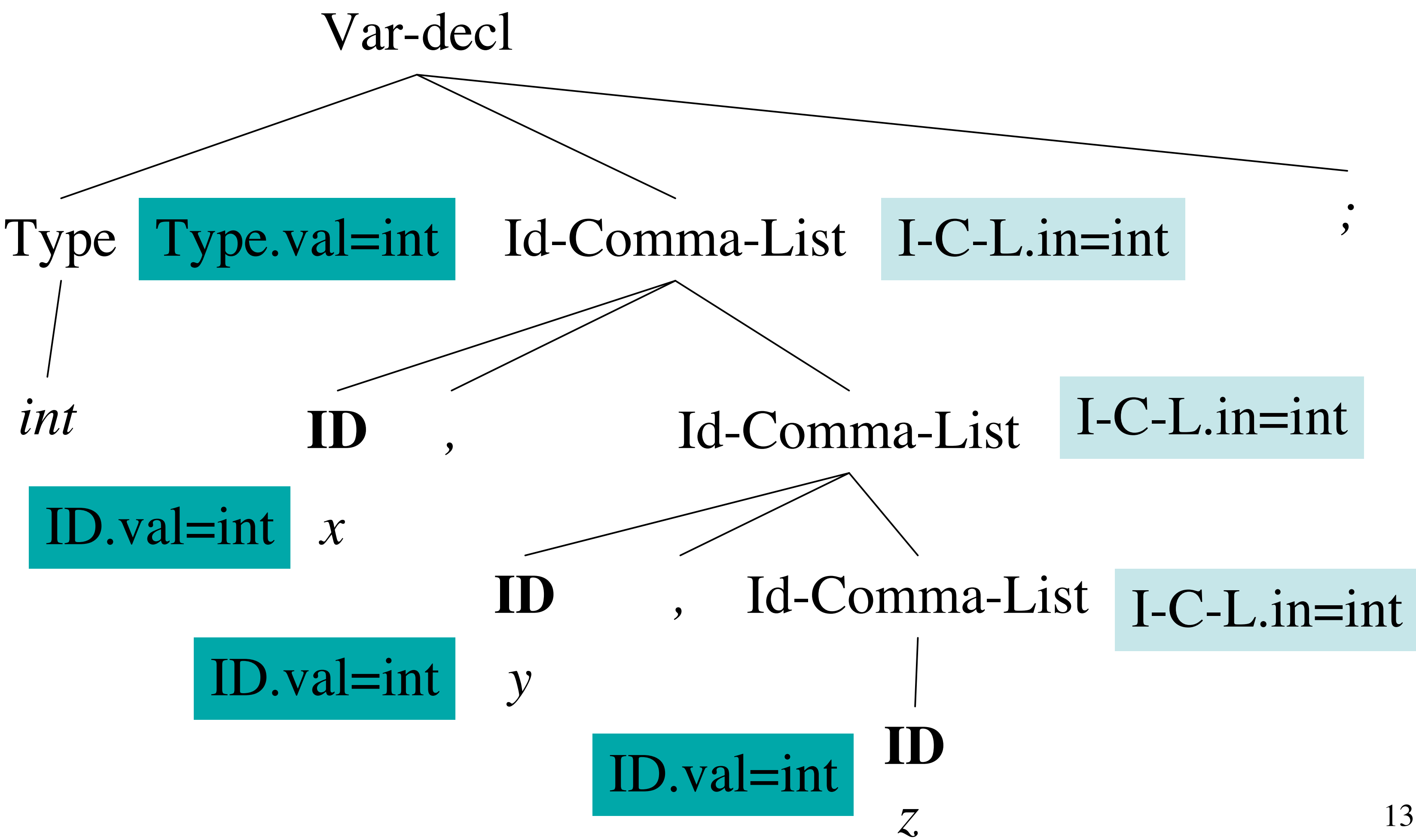
  Id-comma-list → **ID**

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

Example: *int x, y, z ;*



Example: *int x, y, z ;*



Syntax-directed definition

Var-decl → Type Id-comma-list **;**

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

Type → **int | bool**

{ \$0.val = int; } & { \$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; }

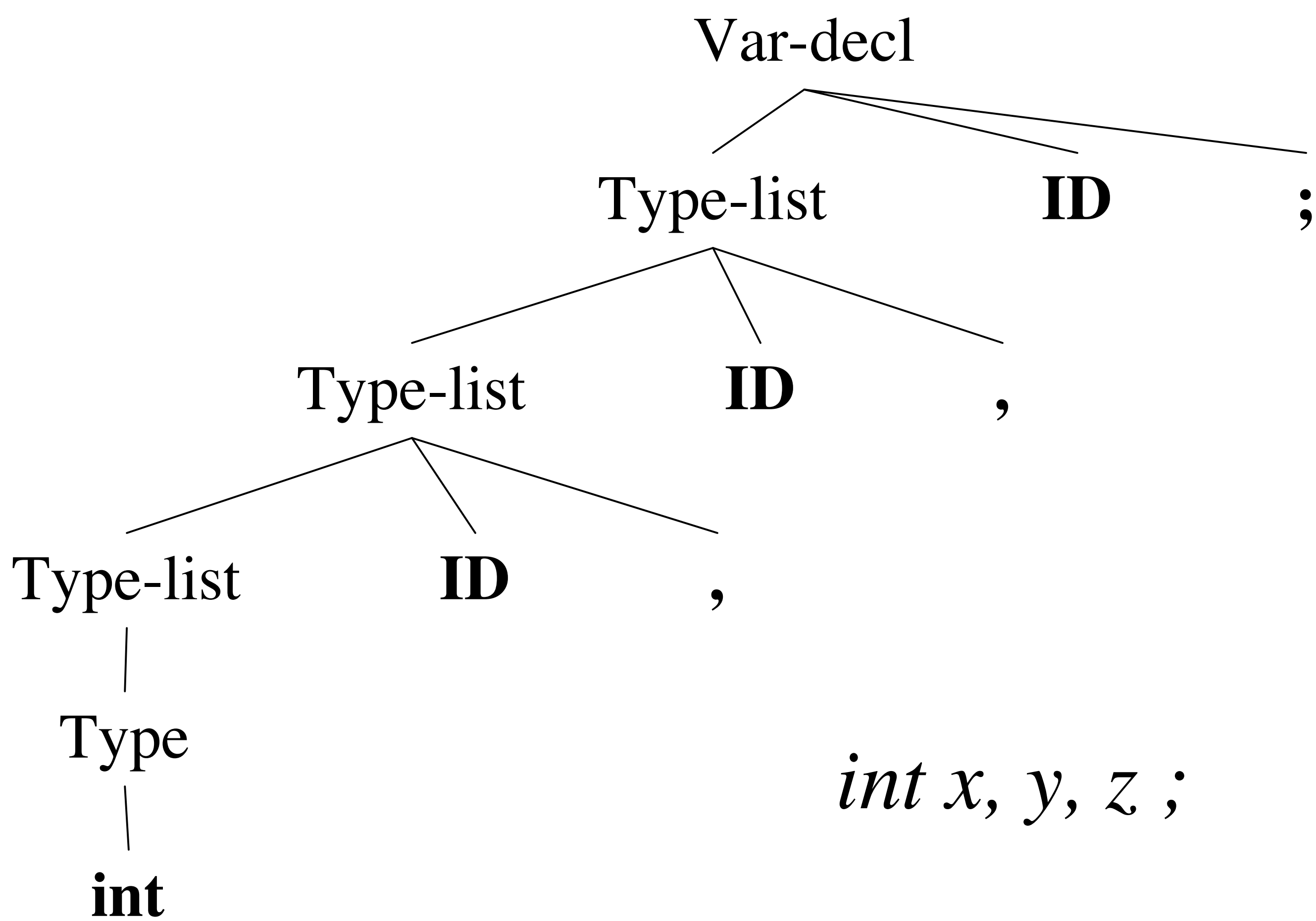
Flow of Attributes in *Var-decl*

- 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 value from its left sibling *Type*
- Computing attributes purely bottom-up is not sufficient in this case
- Do we need synthesized attributes in this grammar?

Inherited Attributes

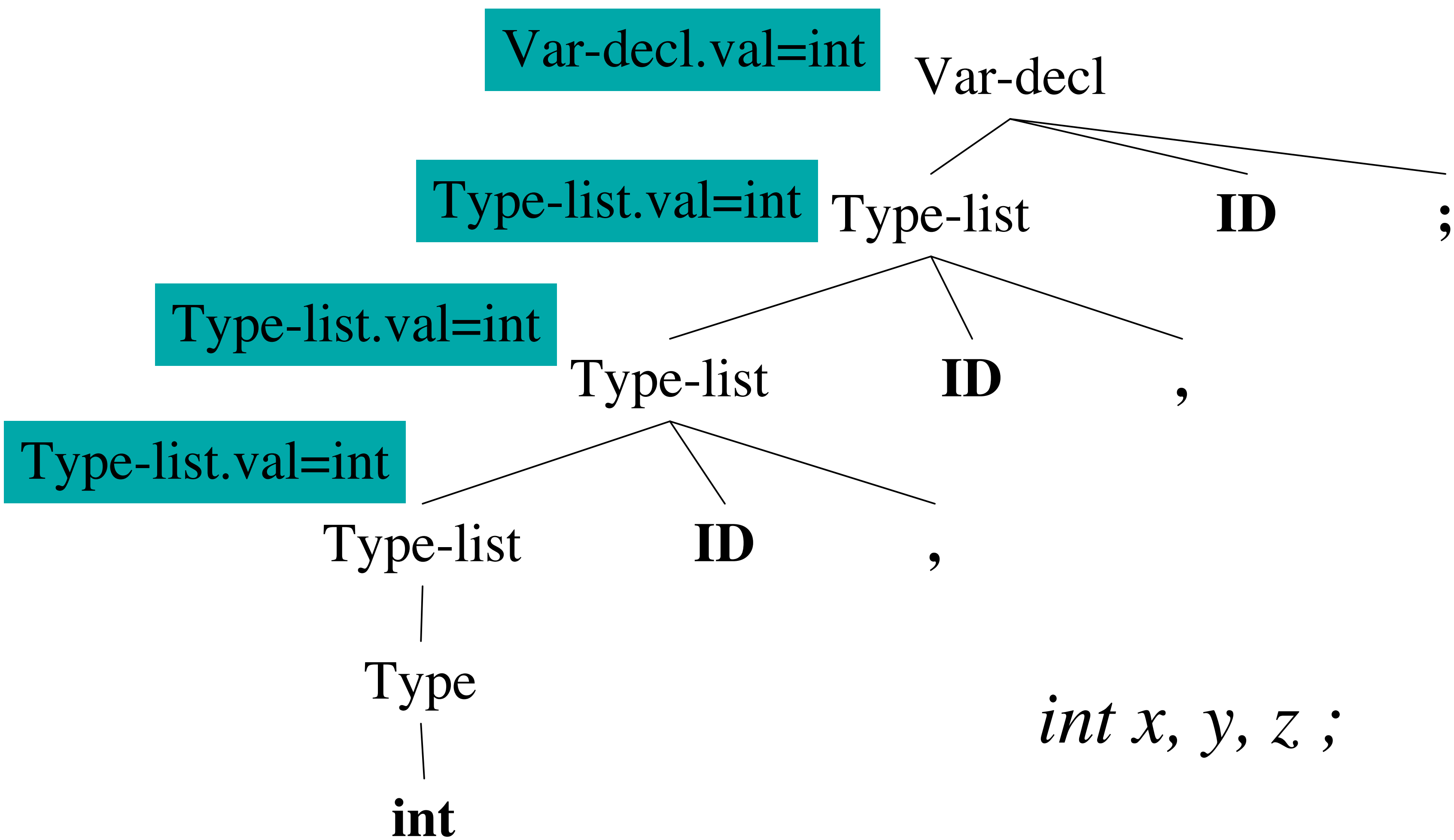
- **Inherited attributes** are attributes that are computed at a node based on attributes from siblings or the parent
- Typically we combine synthesized attributes and inherited attributes
- It is possible to convert the grammar into a form that *only* uses synthesized attributes

Removing Inherited Attributes



int x, y, z ;

Removing Inherited Attributes



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 | bool**

{ $0.val = int; } & { $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 a CFG rule

  $A \rightarrow X_1..X_{j-1}X_j..X_n$ two conditions hold:
  - Each inherited attribute of X_j depends on $X_1..X_{j-1}$
  - Each inherited attribute of X_j depends on A
- These two conditions ensure left to right and depth first parse tree construction
- Every S-attributed definition is L-attributed

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
- Can we still eliminate left-recursion?

Top-down translation

$E \rightarrow E + T$

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

$E \rightarrow E - T$

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

$T \rightarrow$ IntConstant

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

$E \rightarrow T$

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

$T \rightarrow (E)$

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

Top-down translation

$E \rightarrow T\ R$

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

$R \rightarrow +\ T\ R$

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

$R \rightarrow -\ T\ R$

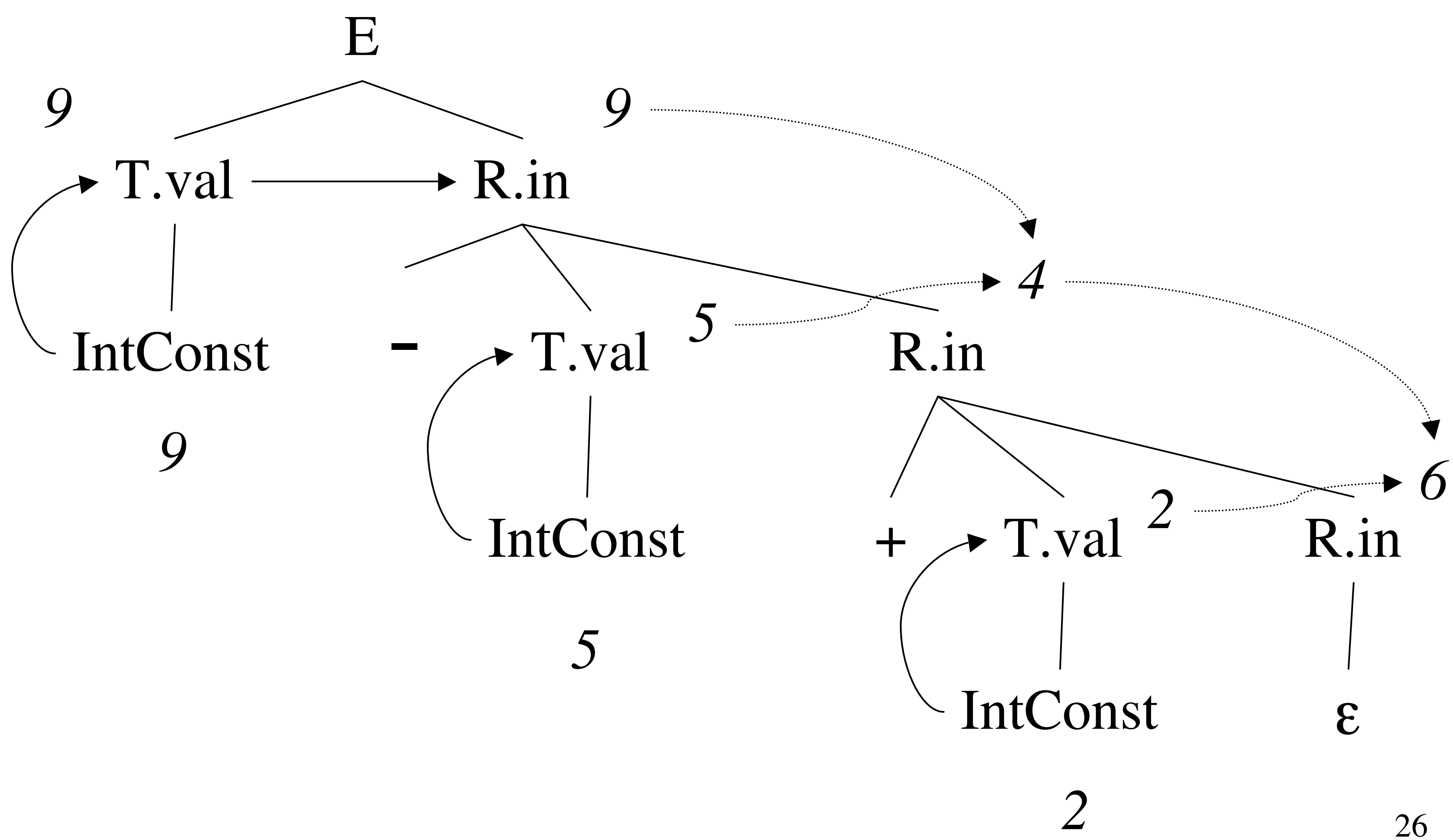
{ \$3.in = \$0.in - \$2.val; \$0.val = \$3.val; }

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

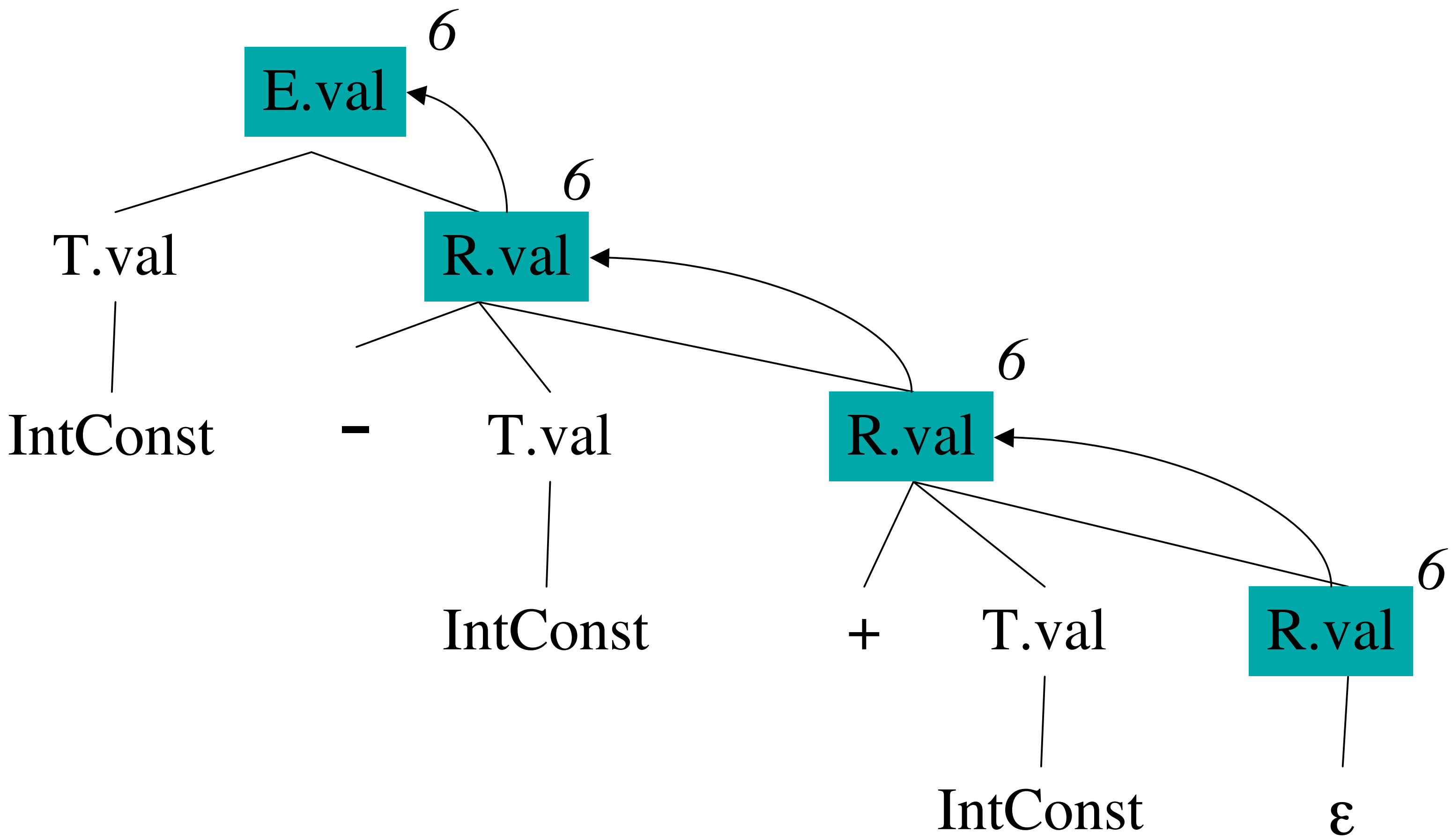
$T \rightarrow (\ E\ )$ { \$0.val = \$1.val; }

$T \rightarrow$ IntConstant { \$0.val = \$1.lexval; }

Example: *9 - 5 + 2*


E
9
9
T.val
R.in
4
IntConst
-
T.val
5
R.in
9
6
IntConst
+
T.val
2
R.in
5
IntConst
ε
2

Example: *9 - 5 + 2*



6
E.val
6
T.val
R.val
6
IntConst
-
T.val
R.val
6
IntConst
+
T.val
R.val
IntConst
ε

Translation Scheme

- A *translation scheme* is a CFG where each rule is associated with a semantic attribute
- A TS that maps infix expressions to postfix:

E → T R

R → + T { print(‘+’); } R

R → - T { print(‘-’); } R

R → ε

T → **id** { print(**id**.lookup); }

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**

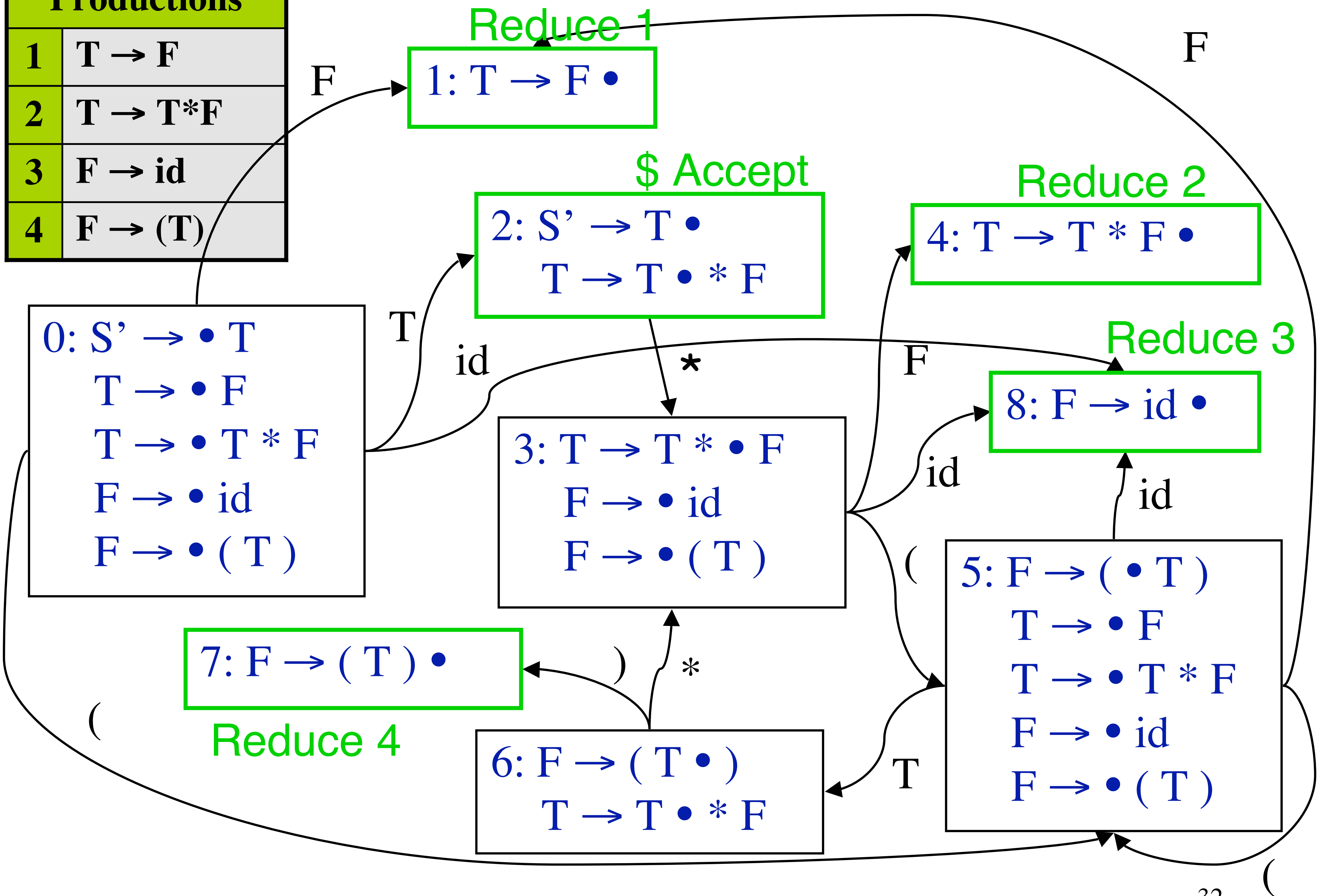
{ val := **id**.lookup();

if (val) { $0.val = $1.val; }

else { error; } }

F → (T) { $0.val = $1.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 } **a.Pop; a.Push 3;**
0 5 6	**) * id $**	**Shift 7**	
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 * $2.val; } **3 pops;** **a.Push 3*2=6**
0 2	**$**	**Accept**	

Example: Inherited Attributes

$E \rightarrow T\ R$

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

$R \rightarrow +\ T\ R$

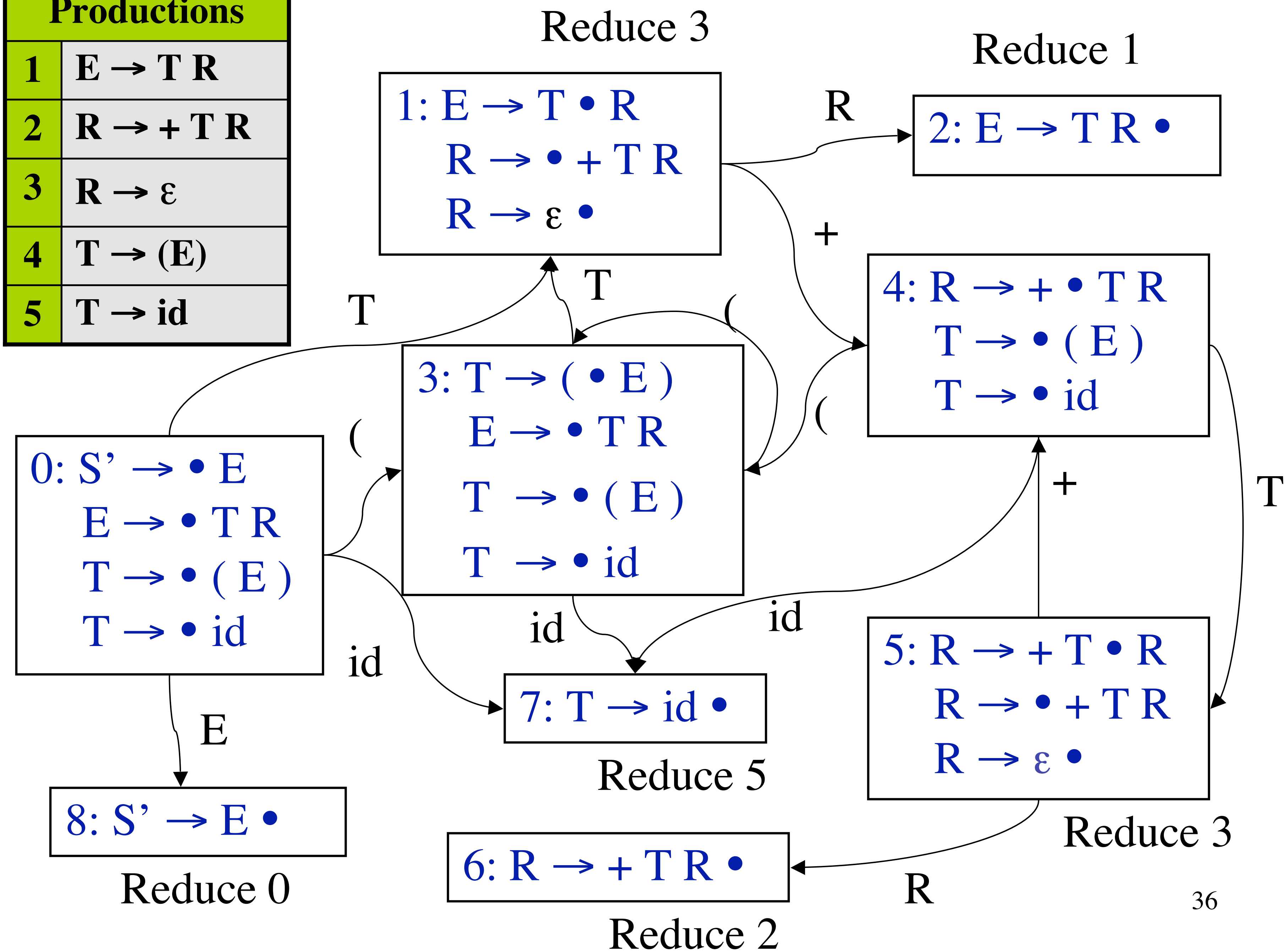
{ \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; }

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

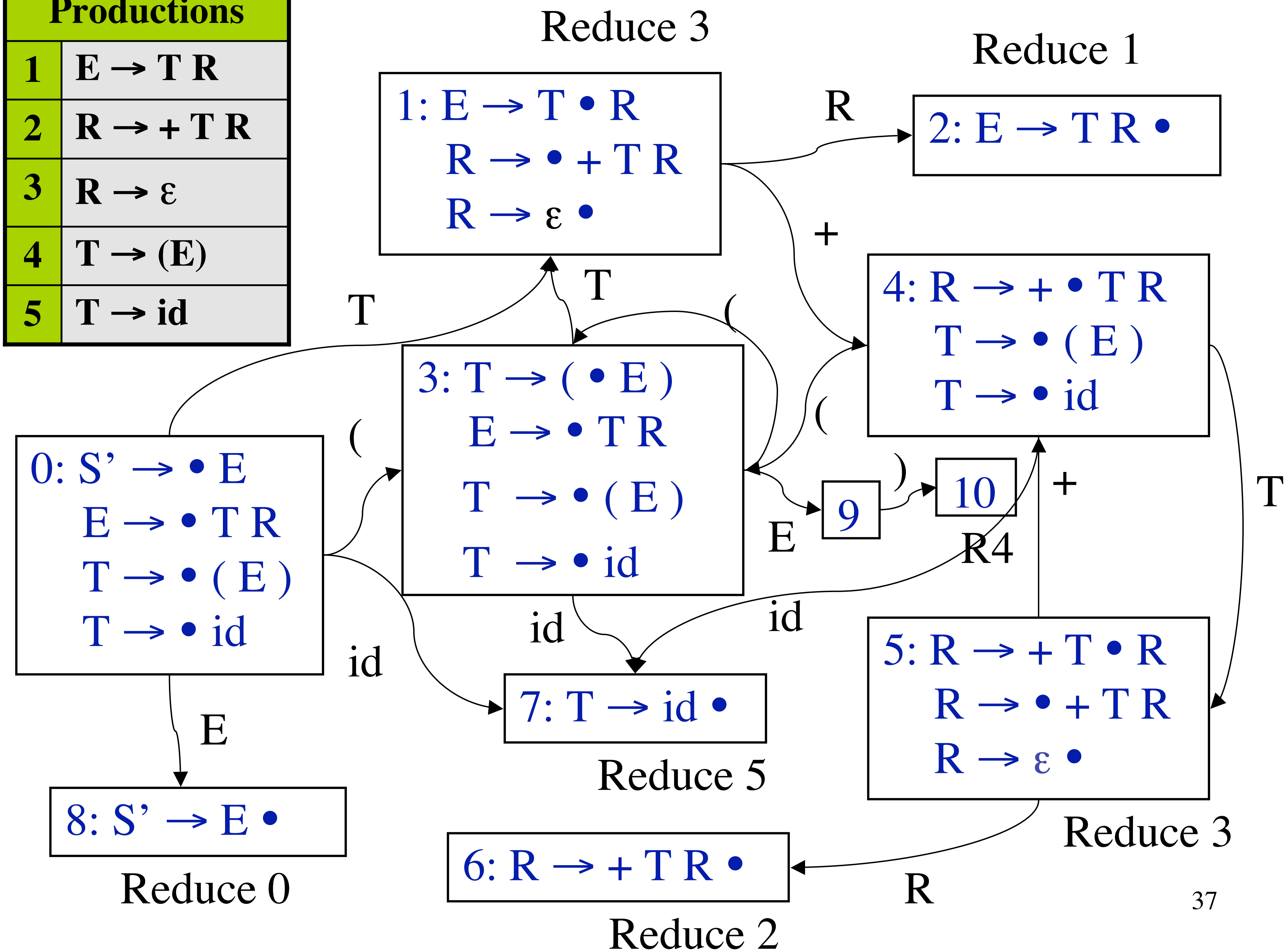
$T \rightarrow (\ E\ )$ { \$0.val = \$1.val; }

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

Marker Non-terminals

E → T R

R → + T { print('+'); } R

R → - T { print('-'); } R

R → ε

T → **id** { print(**id**.lookup); }

Actions that should be done after recognizing T but before predicting R

Marker Non-terminals

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

Equivalent SDT using *marker non-terminals*

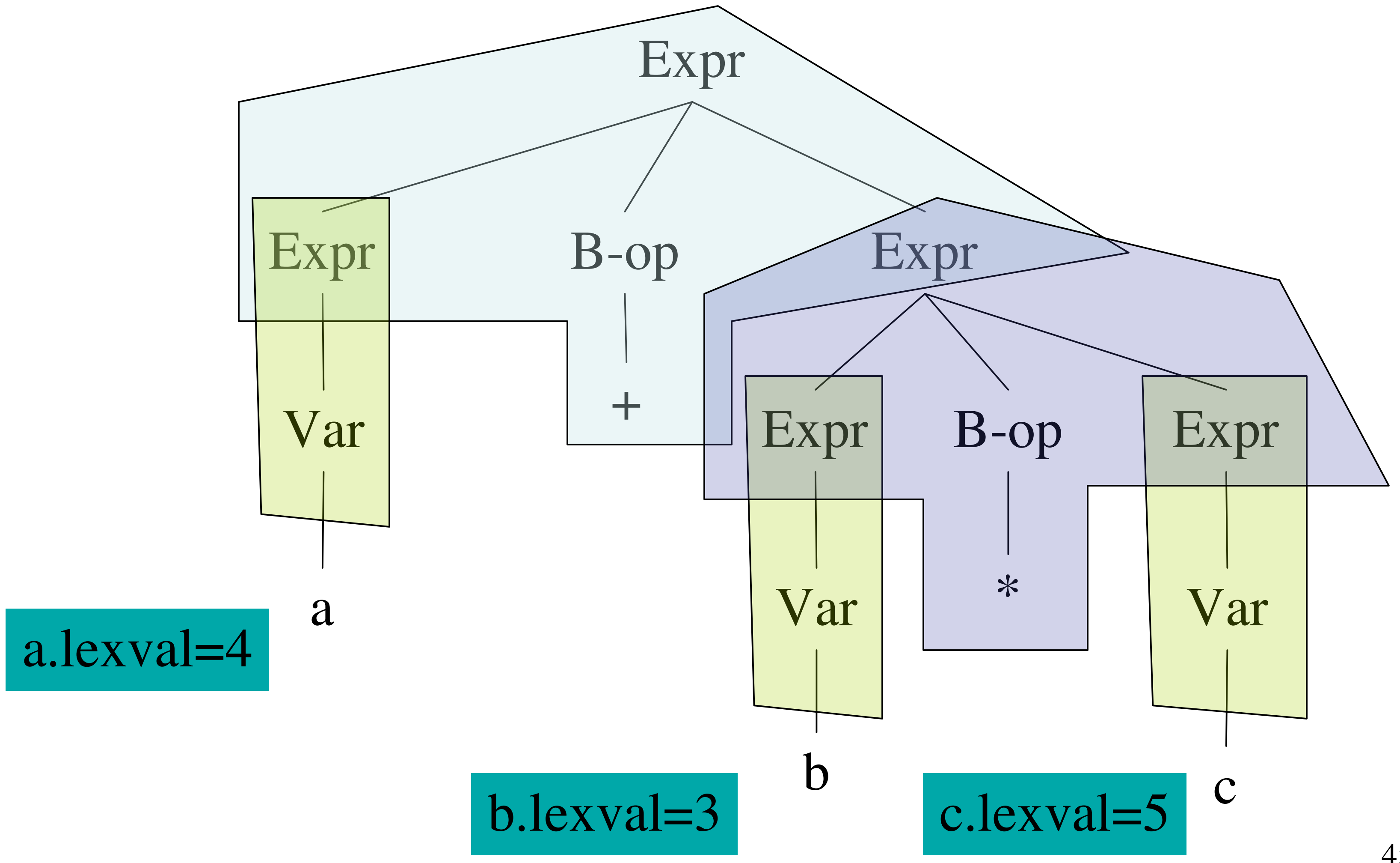
Tree Matching Code Generators

- Write tree patterns that match portions of the parse tree
- Each tree pattern can be associated with an action (just like attribute grammars)
- There can be multiple combinations of tree patterns that match the input parse tree

Tree Matching Code Generators

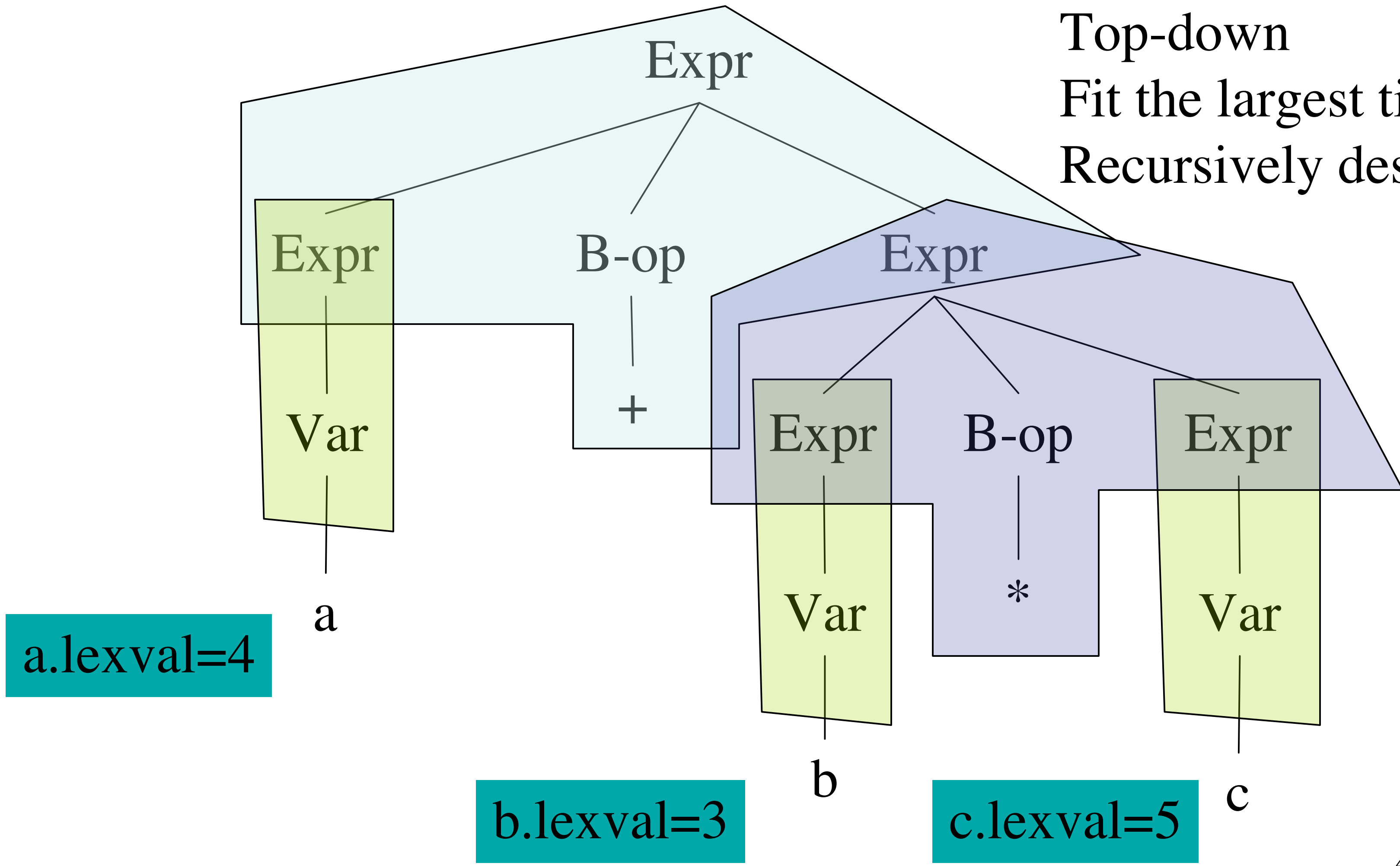
- To provide a unique output, we assign costs to the use of each tree pattern
- E.g. assigning uniform costs leads to smaller code or instruction costs can be used for optimizing code generation
- Three algorithms: Maximal Munch (§9.12), Dynamic Programming (§9.11), Tree Grammars

Maximal Munch: Example 1

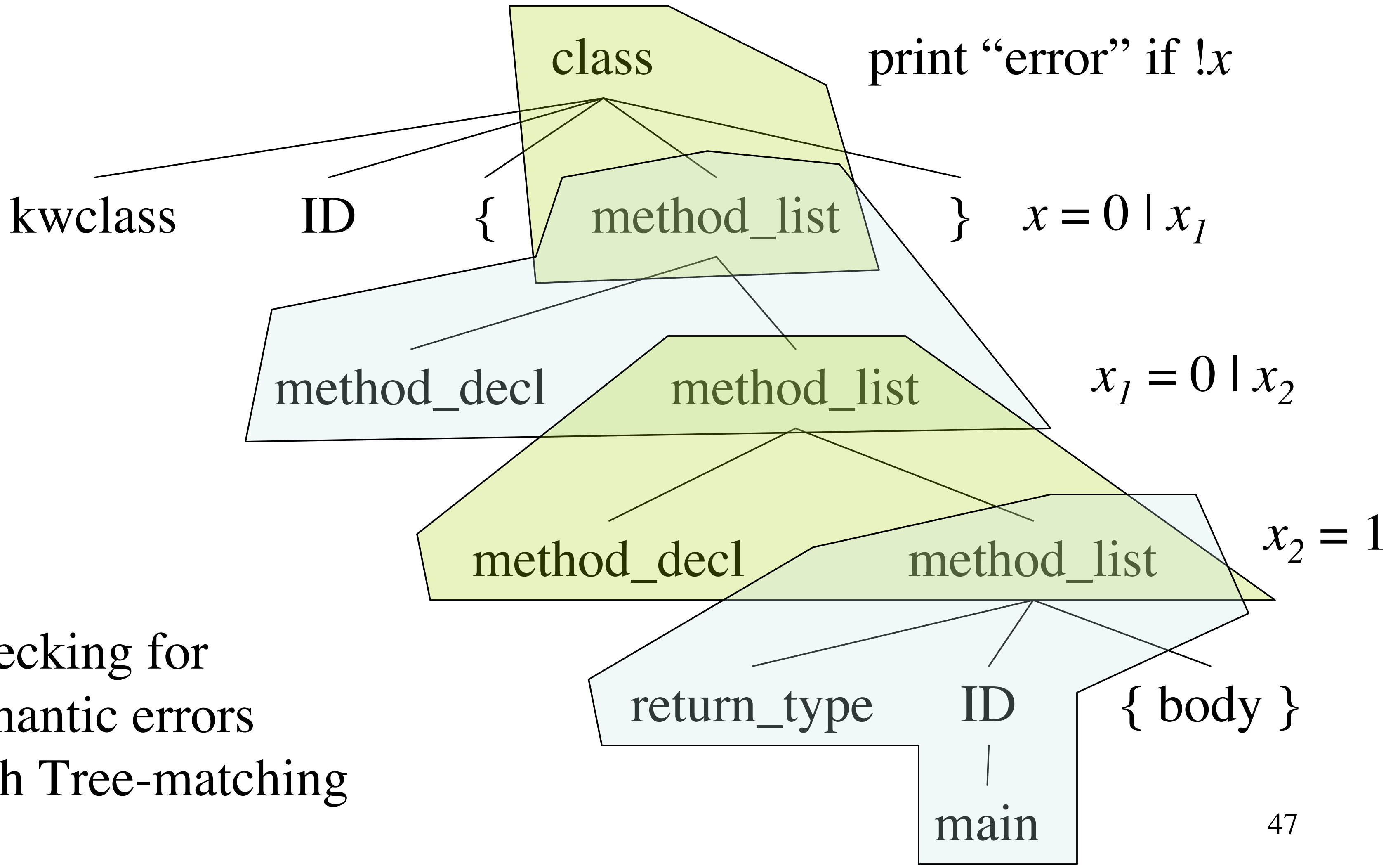


Maximal Munch: Example 1

Top-down
Fit the largest tile
Recursively descend



Maximal Munch: Example 2



Checking for semantic errors with Tree-matching

Tree Parsing Code Generators

- Take the prefix representation of the syntax tree
  - E.g. (+ (* c1 r1) (+ ma c2)) in prefix representation uses an inorder traversal to get + * c1 r1 + ma c2
- Write CFG rules that match substrings of the above representation and non-terminals are registers or memory locations
- Each matching rule produces some predefined output
- Example 9.18 (Dragon book)

Code-generation Generators

- A CGG is like a compiler-compiler: write down a description and generate code for it
- Code generation by:
  - Adding semantic actions to the original CFG and each action is executed while parsing, e.g. yacc
  - Tree Rewriting: match a tree and commit an action, e.g. lcc
  - Tree Parsing: use a grammar that generates trees (not strings), e.g. twig, burs, iburg

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