CMPT 379
Compilers

Anoop Sarkar

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

Parse trees

- Given an input program, we convert the text into a parse tree
- Moving to the backend of the compiler: we will produce intermediate code from the parse tree
- This process is called syntax directed translation because we are using a CFG
- Parser output is a *concrete syntax tree*

Intermediate Representations

- A parse tree is an example of a very high level intermediate representation
- We can reconstruct the original source code from the concrete syntax tree
- Typically we want to check some semantic rules on the parse tree and report any errors
- The next step: semantic processing and code generation

Abstract Syntax Trees

- Take the concrete syntax tree and simplify it to the essential nodes
- For example, if the parser used an LL(1) grammar then the concrete syntax tree will have extra non-terminals
- Elimination of left-recursion, changing the grammar to remove shift/reduce conflicts

Abstract Syntax Trees

- Other examples include lists of various kinds that involves recursion in CFGs:

  Program → Function-List

  Function-List → Function-Defn Function_List

  | Function-Defn

- The extra nodes created due to these grammar changes are not useful
- The extra nodes might make things non-local (inconvenient) for the semantic processing and code generation

Abstract Syntax Trees

- Process the concrete syntax tree and convert into a tree that is useful for semantic processing and code generation
- Note that ambiguity is no longer a problem: we already have the parse tree
- Abstract syntax trees will typically have pointers to children *and* pointers to parent nodes

Example

- Consider the following fragment of a programming language grammar:

  Program → Function-List

  Function-List → Function-Defn Function-List

  | Function-Defn

  Function-Defn → **fun id** (Param-List) Body

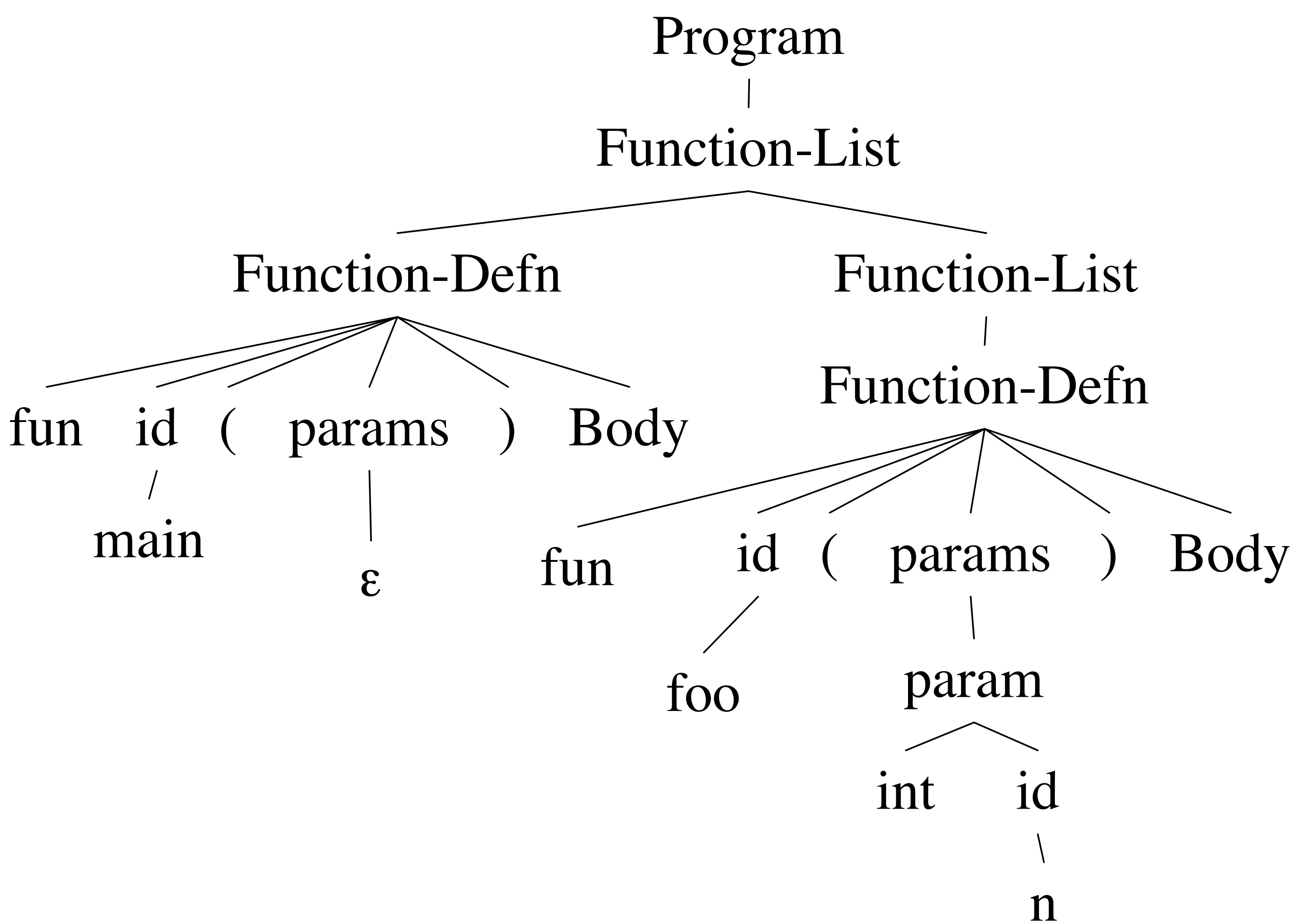
  Body → ‘{‘ Statement-List ‘}’

Example (cont’d)

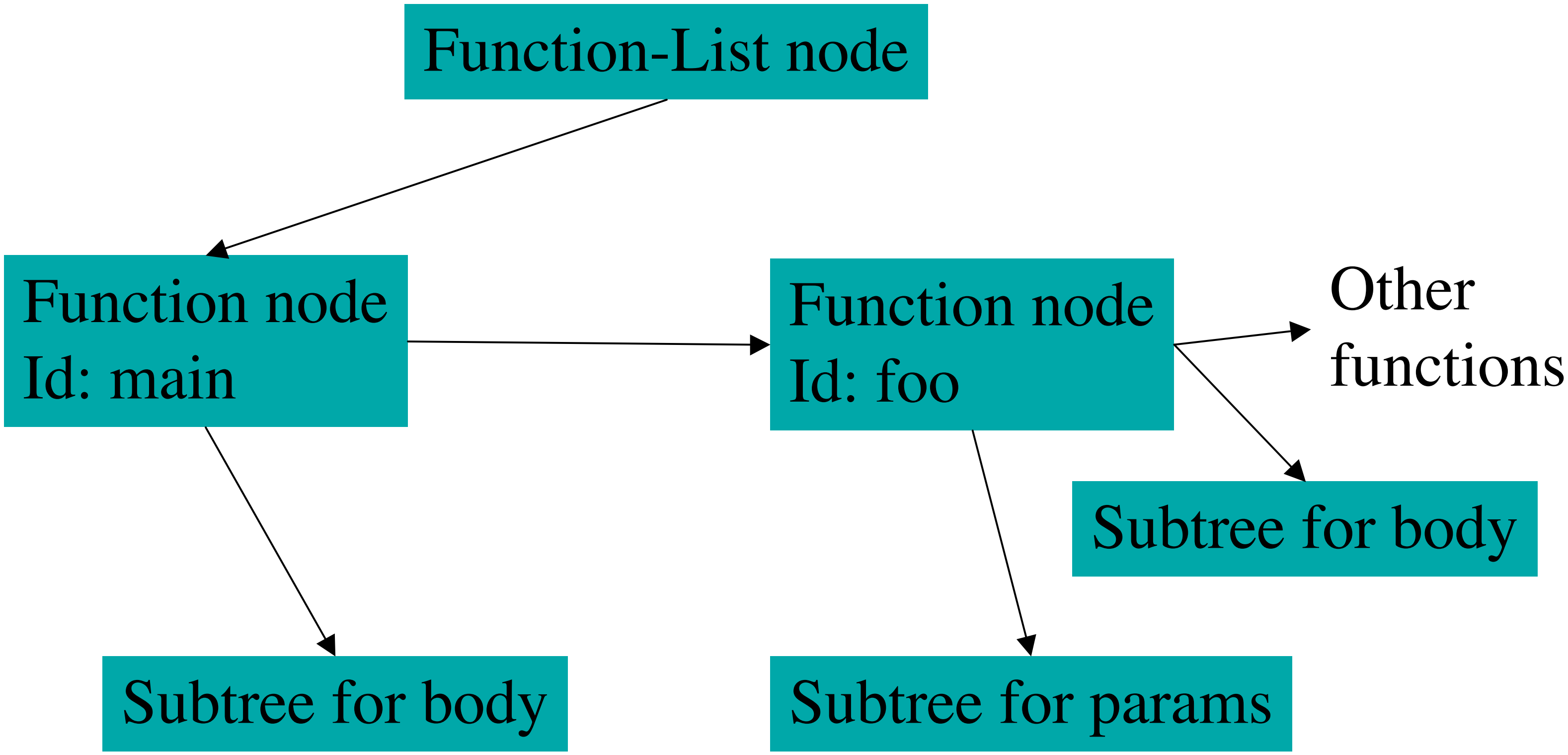
- Consider an example program:

```
fun main ()
{
    statement
}
fun foo (int n)
{
    statement
}
```

Concrete Parse Tree



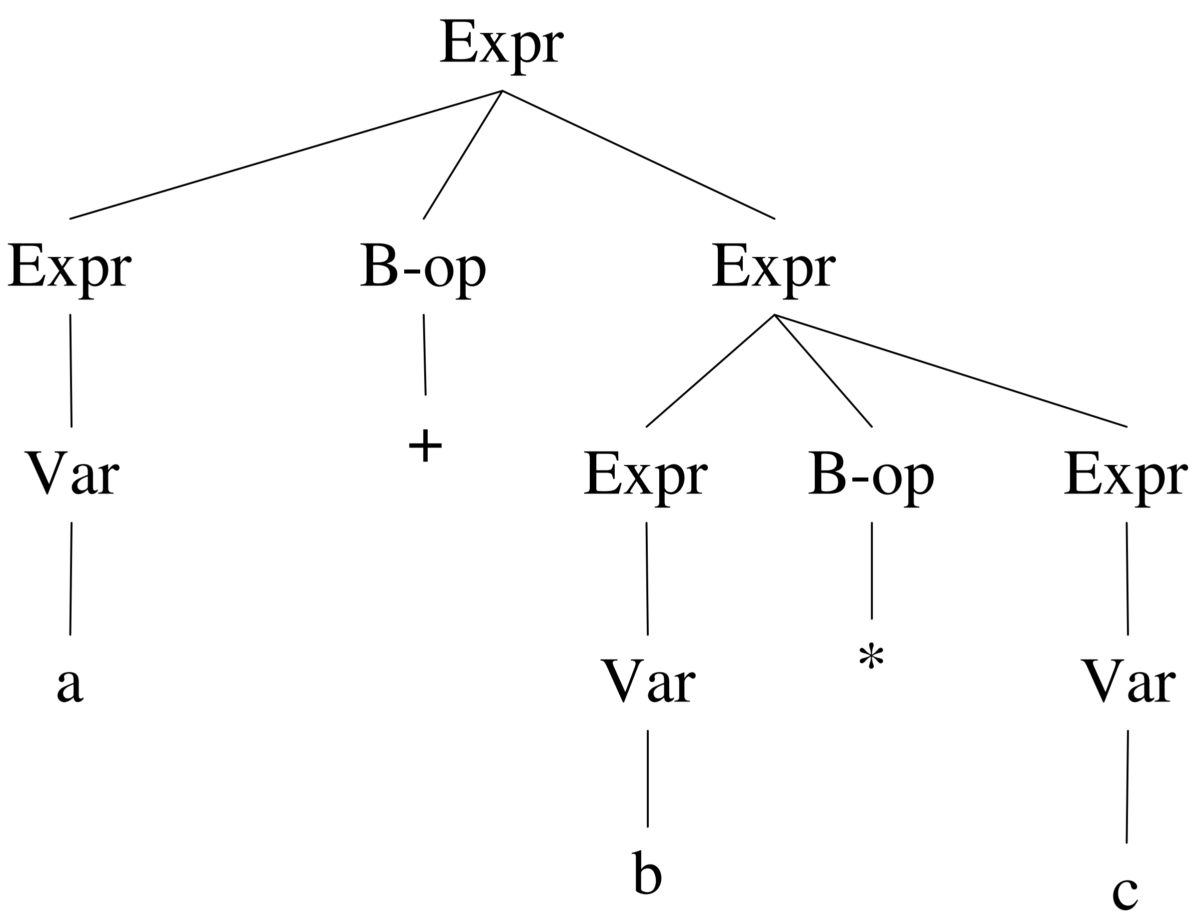
Abstract Parse Tree



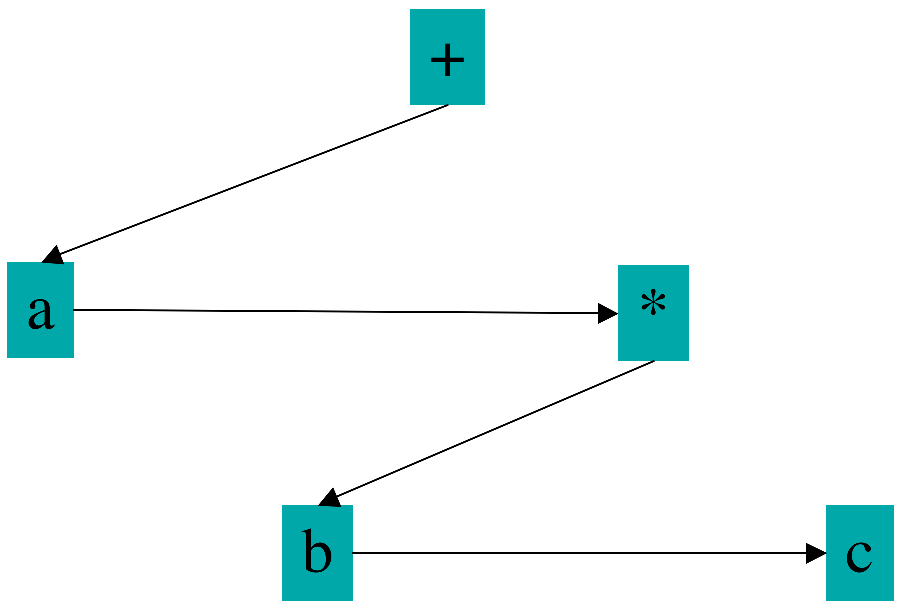
Code generation as Translation

- Code generation can be viewed as translation from the parse tree
- In other words, an alignment between the source code and the assembly code
- Typically we go to an intermediate representation and then to assembly
- Let's consider a simple case where the IR step can be skipped

Expr concrete syntax tree



Expr abstract parse tree



Code generation

- GenerateCode(tree t, int resultRegister)
- Recursively traverse the abstract syntax tree
- At each node produce the code needed for that binary operation based on the results from the recursive call results

Trace of code generation

```
GenerateCode(+, 0)
    GenerateCode(a, 0)
        Write "LOAD a, R0"
    GenerateCode(*, 1)
        GenerateCode(b, 1)
            Write "LOAD b, R1"
        GenerateCode(c, 2)
            Write "LOAD c, R2"
        Write "MUL R1, R2"
    Write "ADD R0, R1"
```

Result of code generation

- The resulting assembly code:

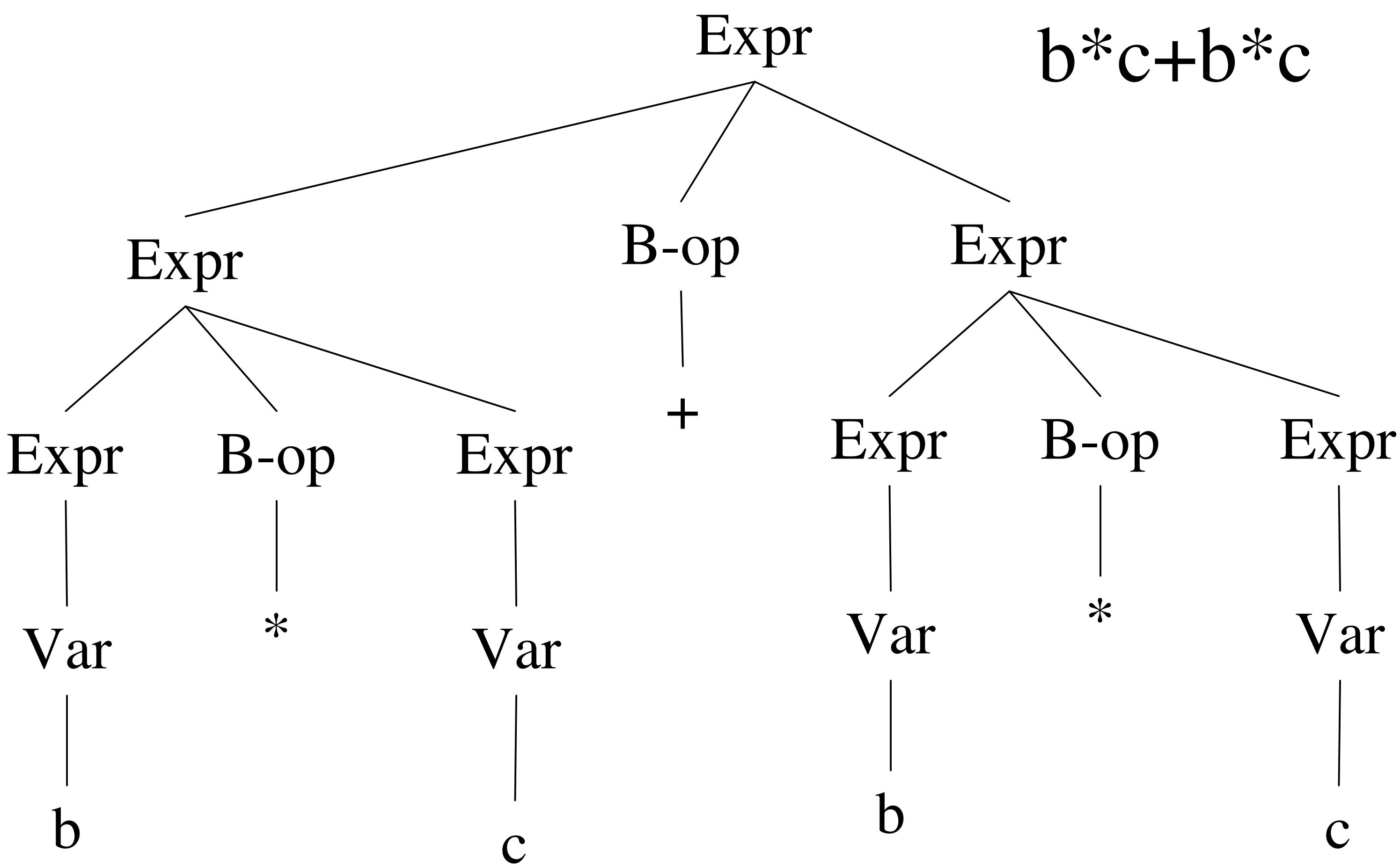
```
LOAD a, R0
LOAD b, R1
LOAD c, R2
MUL R1, R2
ADD R0, R1
```

- Note that using the tree structure means that the registers do not conflict
- Later we will consider the optimal assignment of values to registers

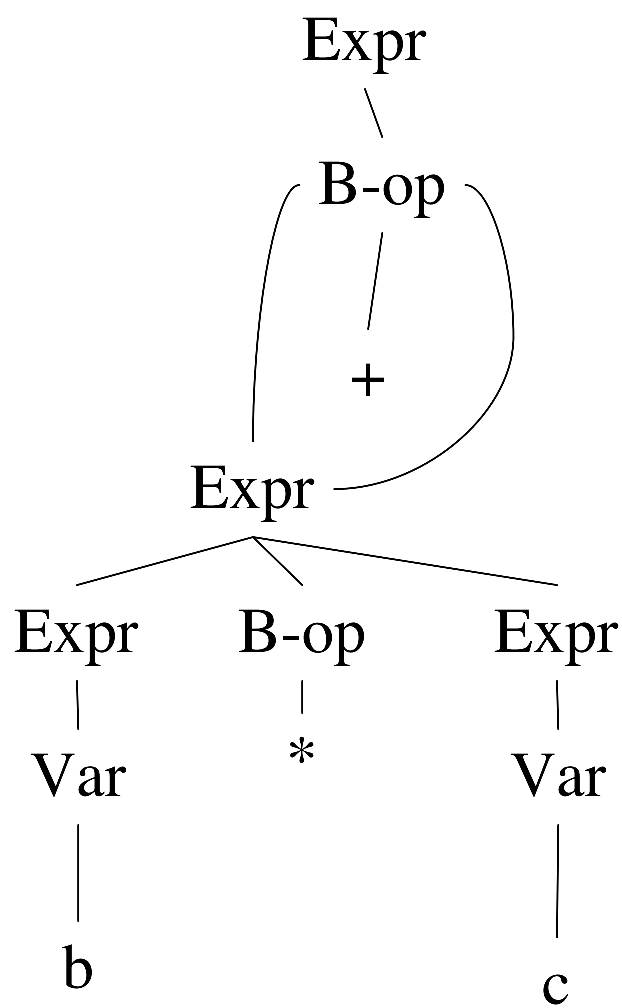
Case Study: Lisp

- The term abstract syntax was coined by John McCarthy
- McCarthy designed Lisp which directly used an abstract syntax bypassing the concrete syntax step
- Structure of Lisp: (*function arg-list*)
- Directly represents the parse tree in syntax
- Lisp: Lots of Irritating Silly Parentheses

Directed Acyclic Graphs



Directed Acyclic Graphs



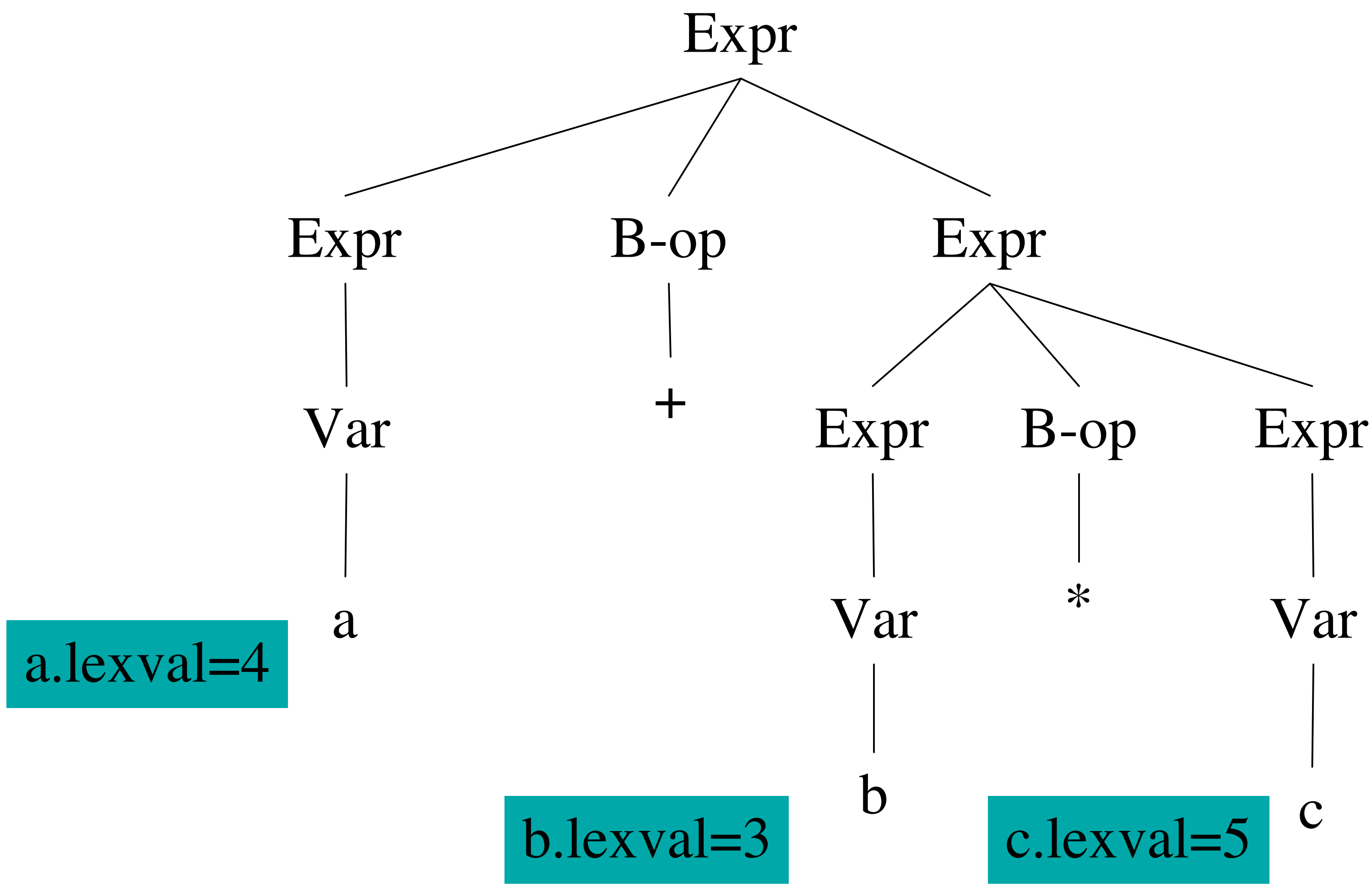
Attribute Grammars

- 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

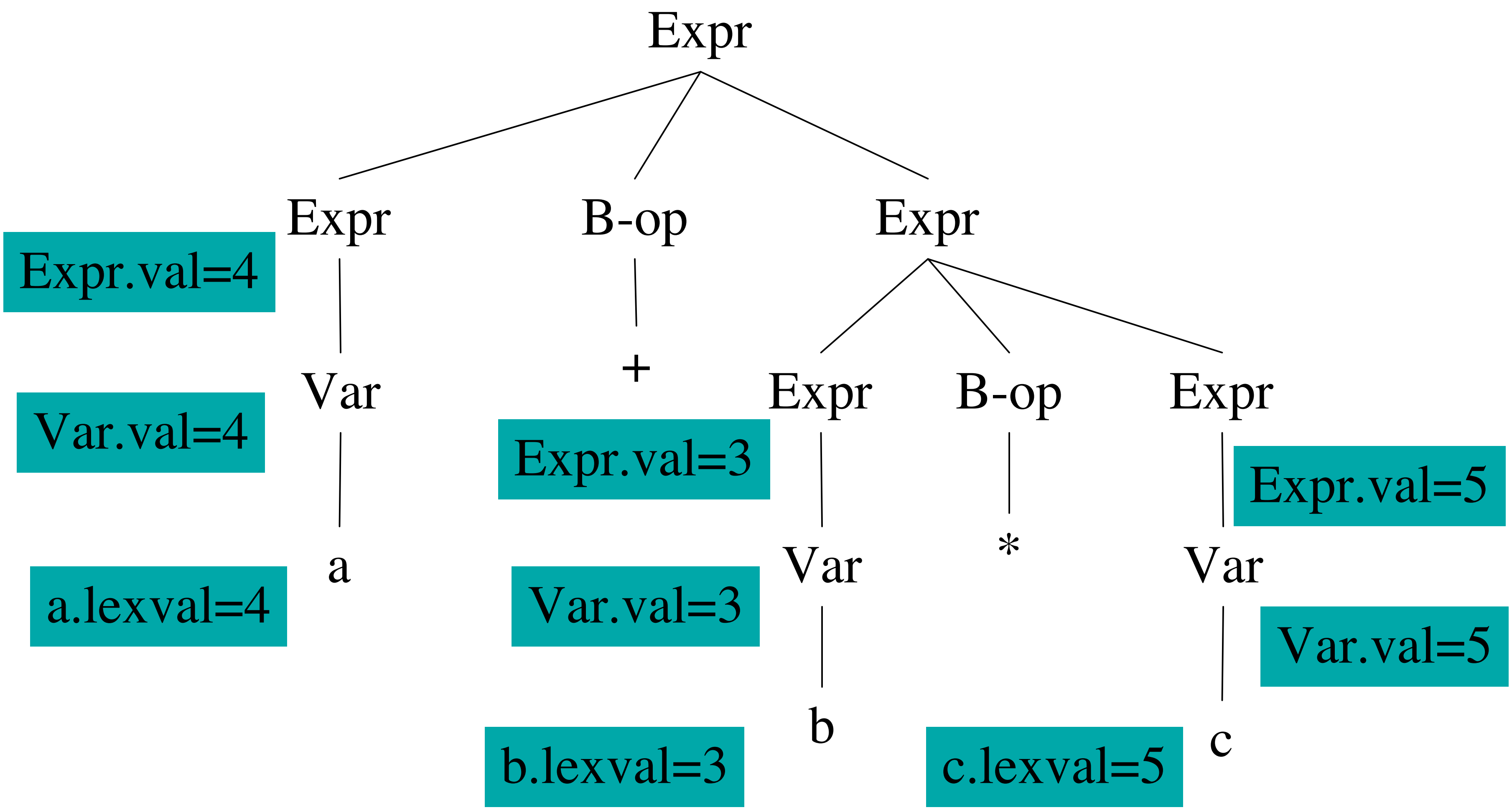
- A CFG can be viewed as a (finite) representation of a function that relates strings to parse trees
- Similarly, an attribute grammar is a way of relating strings with “meanings”
- Since this relation is syntax-directed, we associate each CFG rule with a semantics (rules to build an abstract syntax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

Example

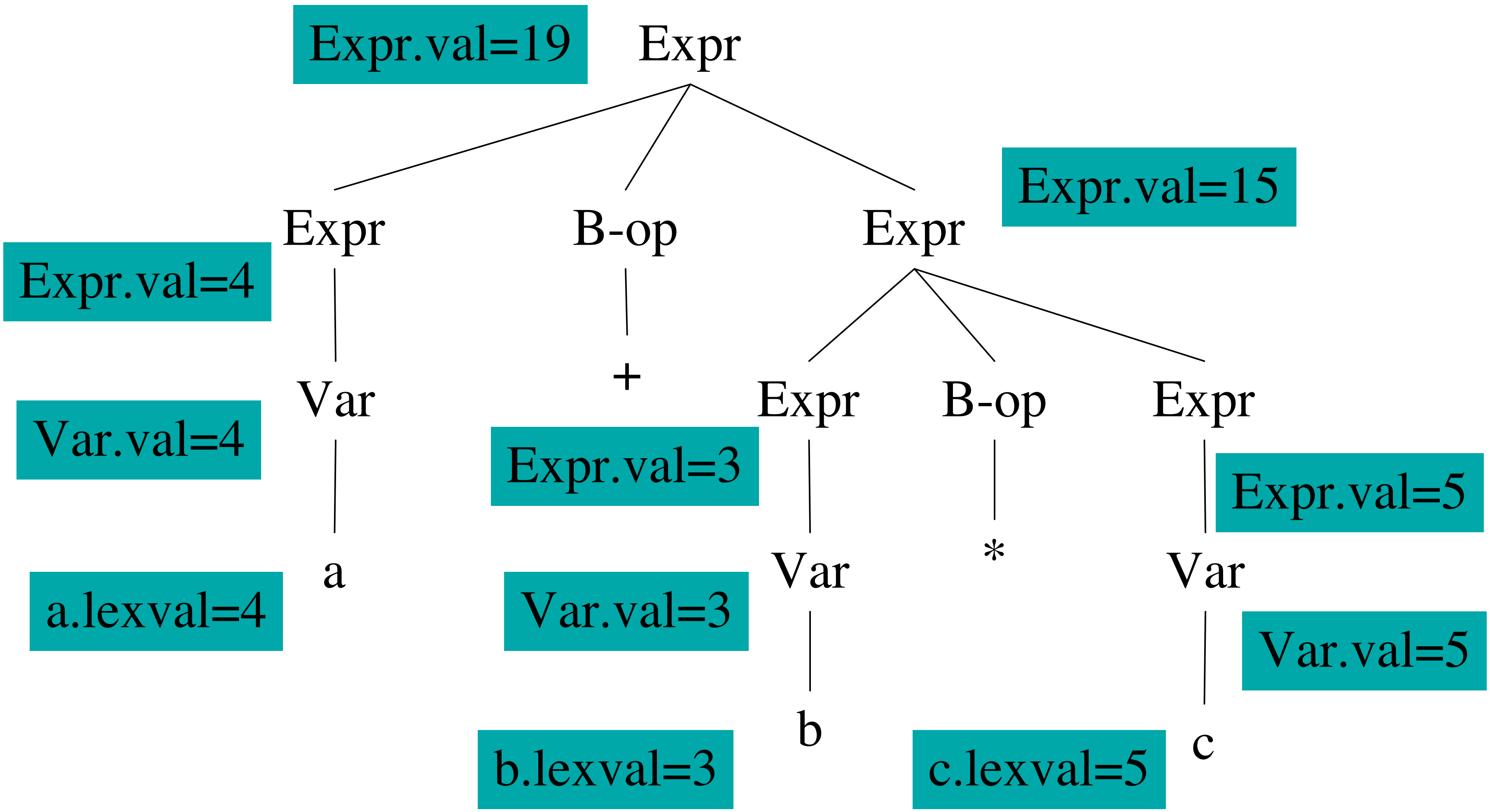


Expr
Expr
B-op
Expr
Var
+
Expr
B-op
Expr
a
a.lexval=4
Var
*
Var
b
b.lexval=3
c.lexval=5
c

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.val=15
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

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:

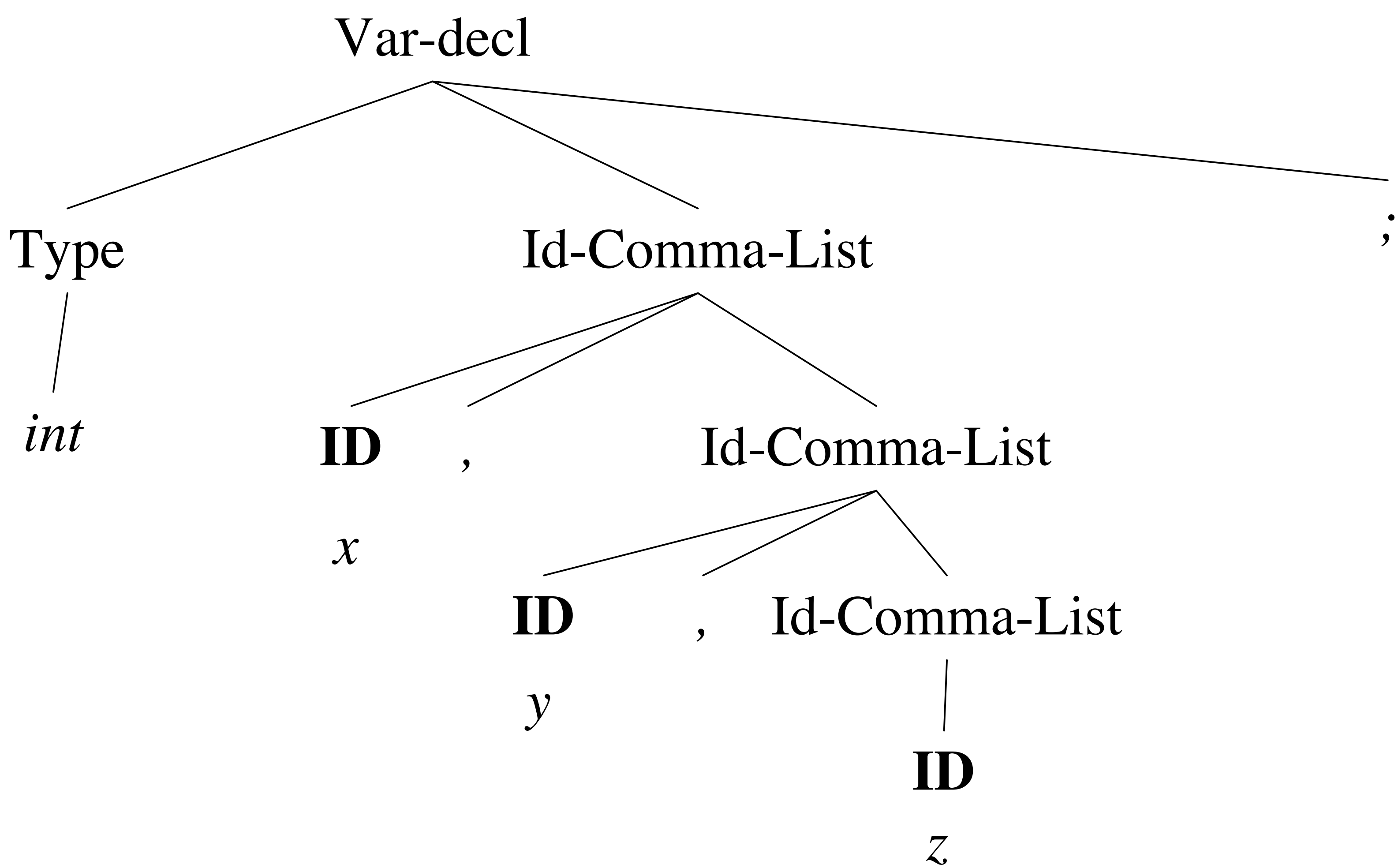
  Var-decl → Type Id-comma-list **;**

  Type → **int** | **bool**

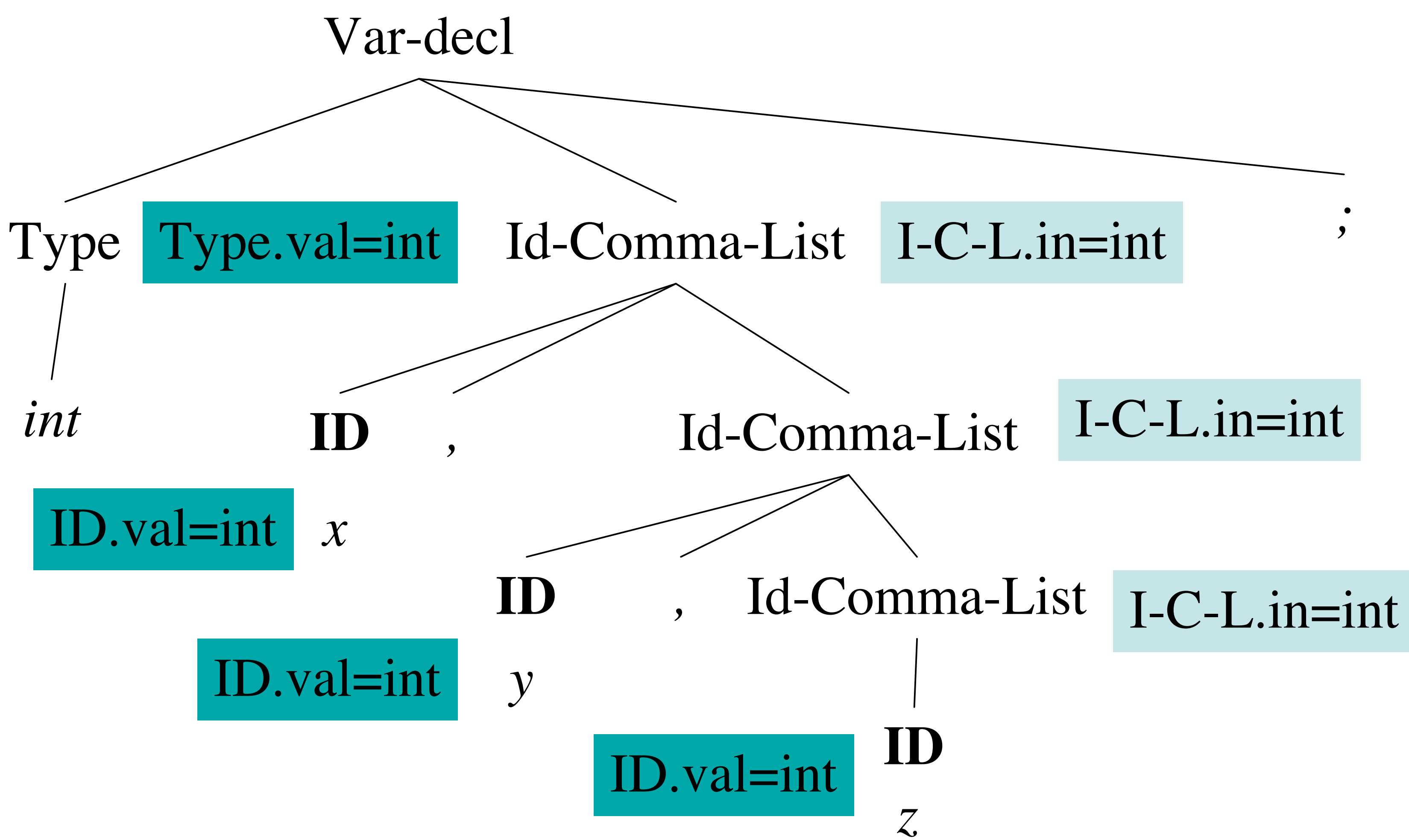
  Id-comma-list → **ID**

  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; }

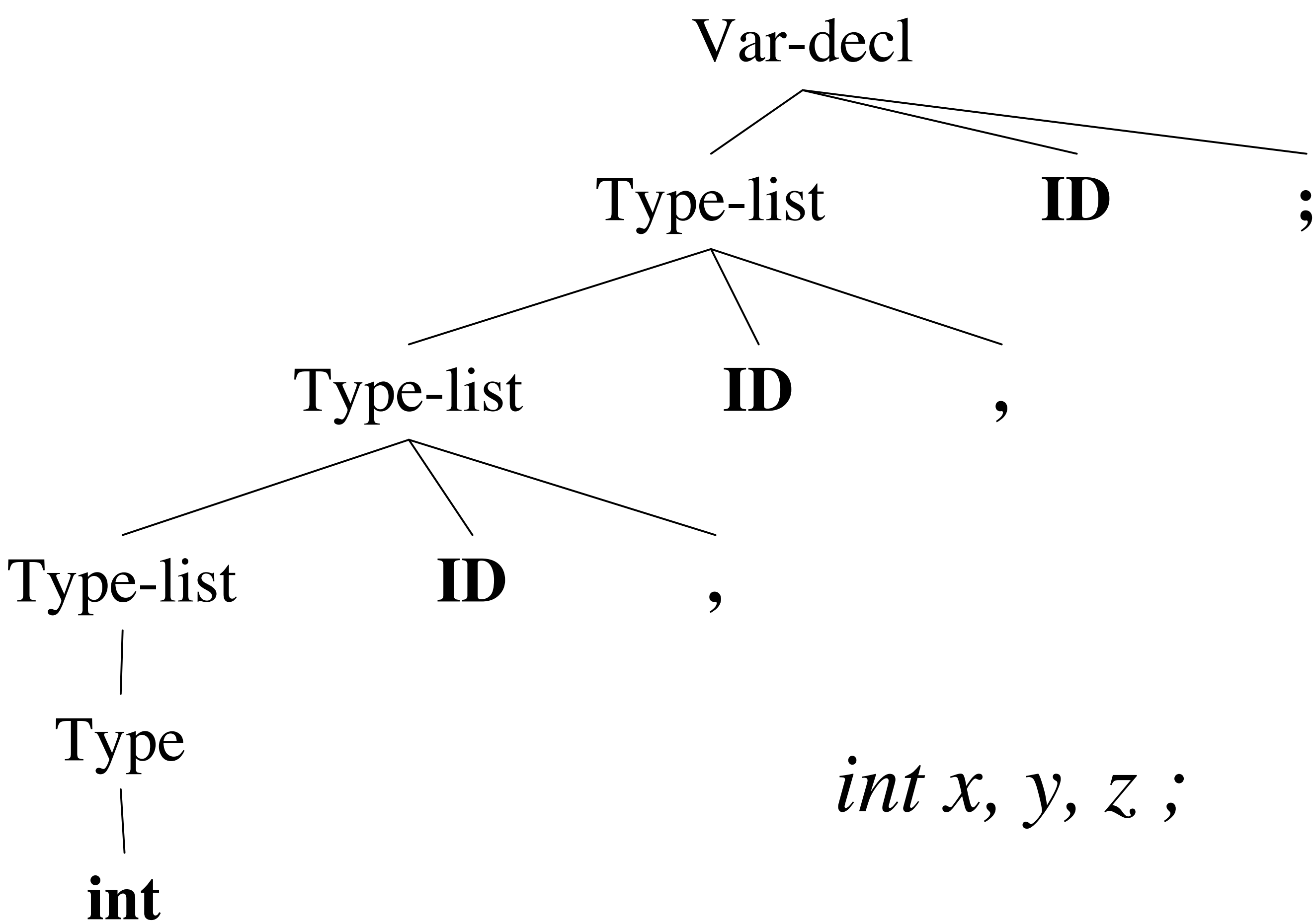
Flow of Attributes in *Var-decl*

- 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

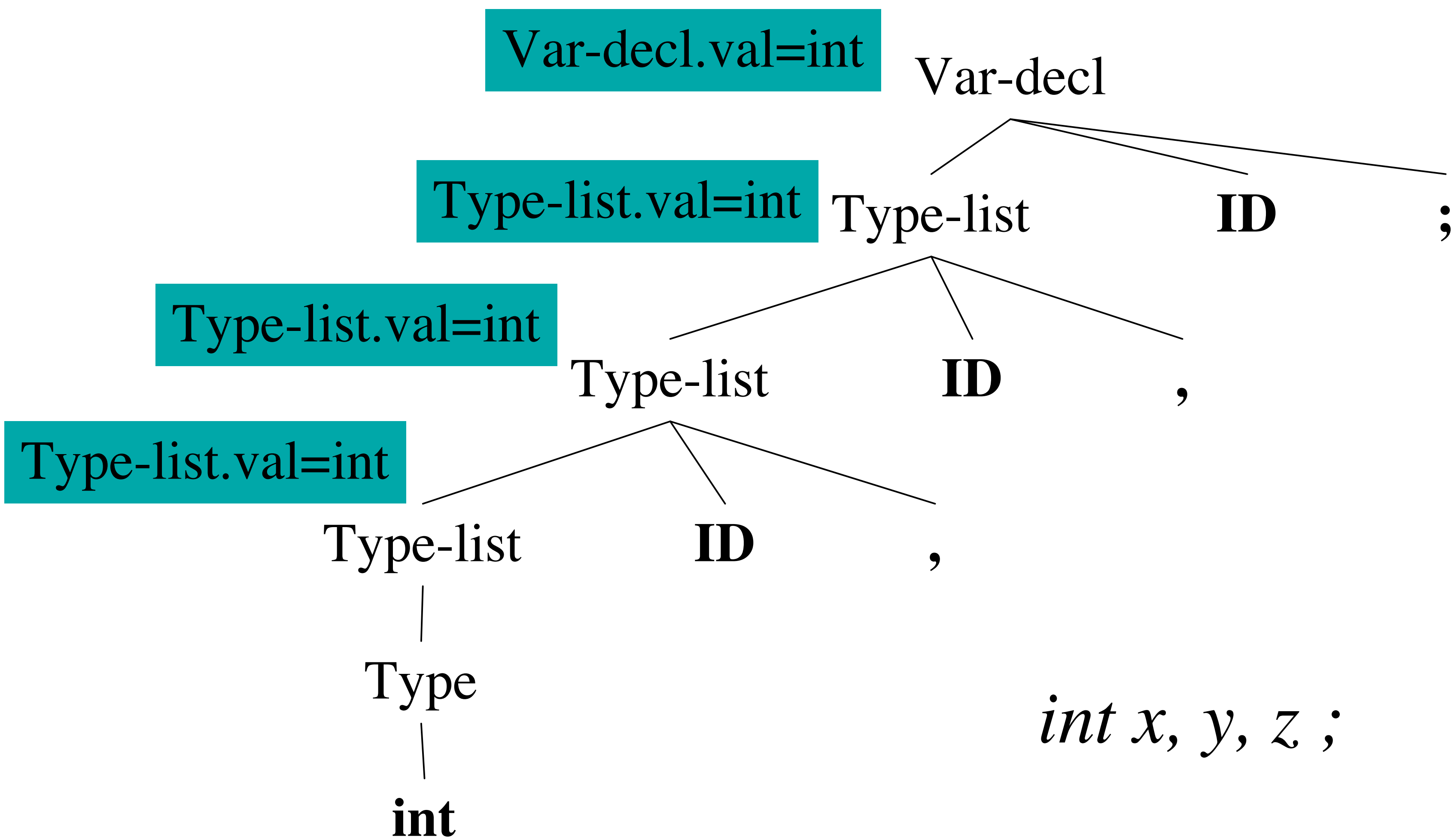
- **Inherited attributes** are attributes that are computed a node based on attributes from siblings or the parent
- Typically we combine synthesized attributes and inherited attributes
- It is possible to convert the grammar into a form that *only* uses synthesized attributes

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 → L M

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

  A → 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; }

$R \rightarrow -\ T\ R$

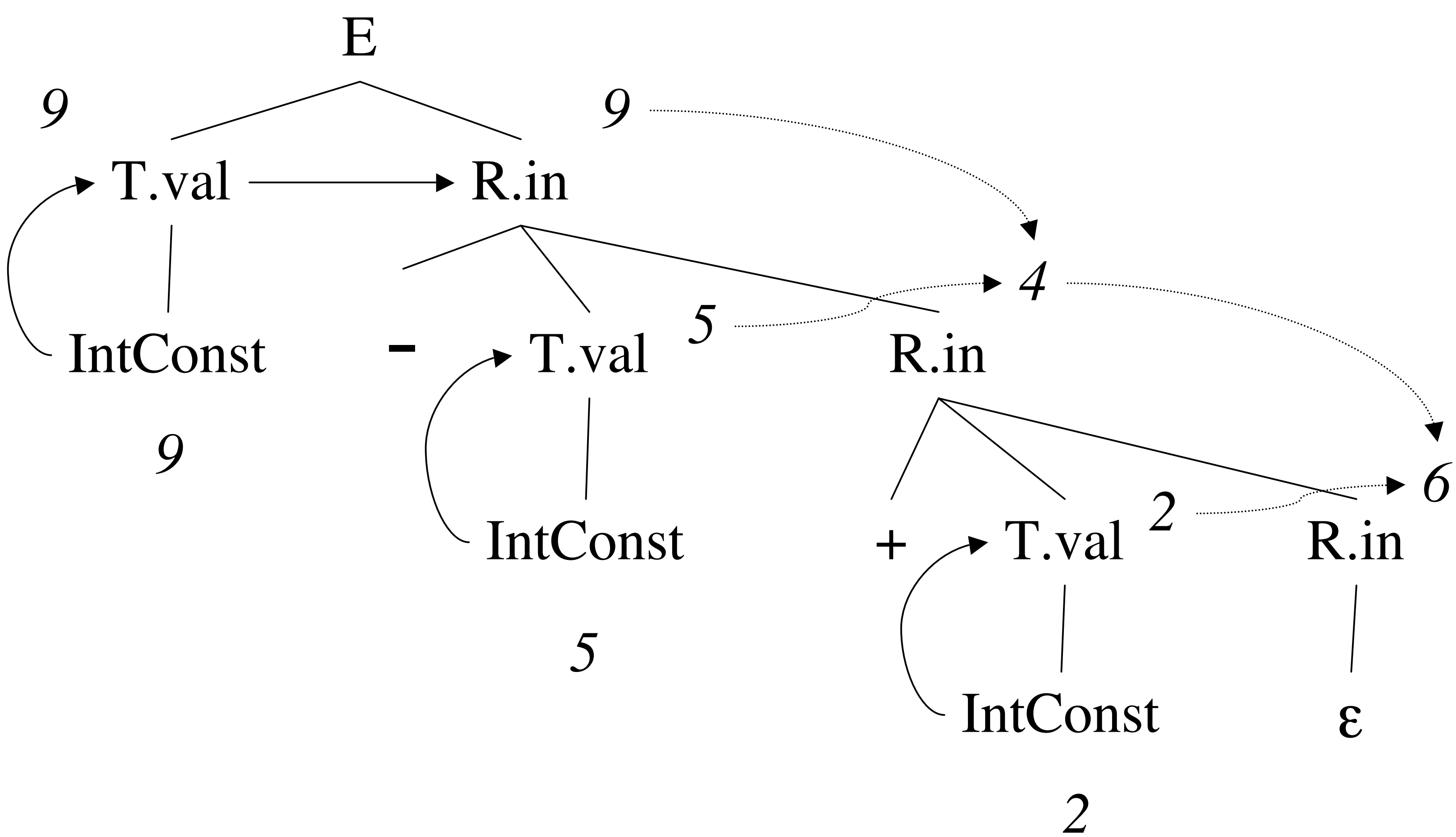
{ \$3.in = \$0.in - \$2.val; \$0.val = \$3.val; }

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

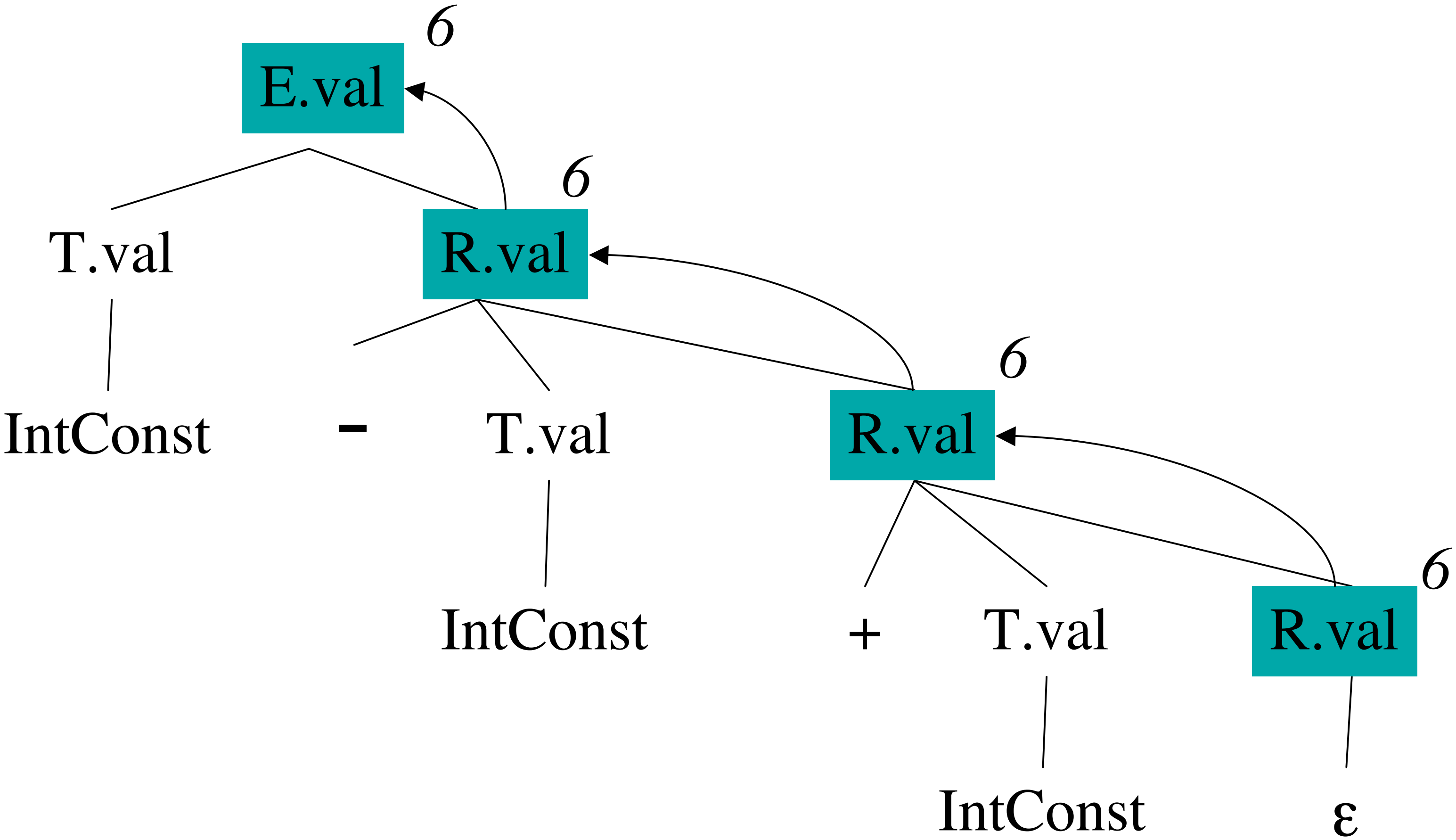
$T \rightarrow (\ E\ )$ { \$0.val = \$1.val; }

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

Example: *9 - 5 + 2*


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

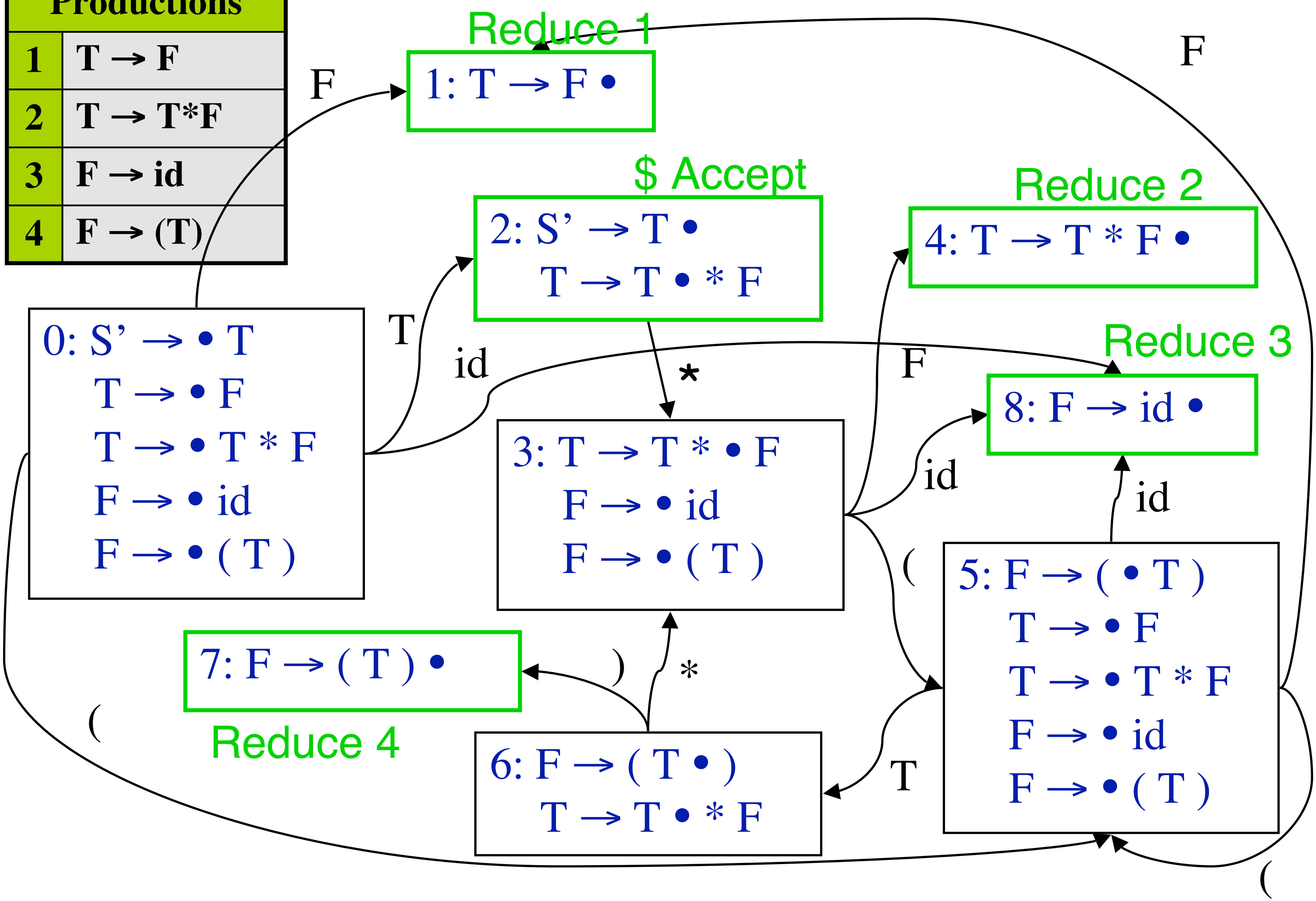
LR parsing and inherited attributes

- 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

	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 S5**	
0 5	**id) * id $**	**Shift S8**	**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 S7**	**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,**	{ 0.val = 1.val }
		pop 1, goto [0,T]=2	**a.Pop; a.Push 3**
0 2	*** id $**	**Shift S3**	**a.Push mul**
0 2 3	**id $**	**Shift S8**	**a.Push id.val=2**
0 2 3 8	**$**	**Reduce 3 F→id,**	
		pop 8, goto [3,F]=4	**a.Pop a.Push 2**
0 2 3 4	**$**	**Reduce 2 T→T * F**	{ 0.val = 2.val(1.val,
		pop 4 3 2, goto [0,T]=2	2.val) }
0 2	**$**	**Accept**	**3 pops;**
			a.Push mul(3,2)=6

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