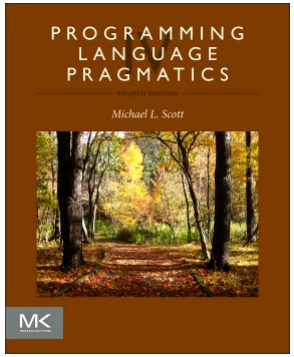
Chapter 4 :: Semantic Analysis

Programming Language Pragmatics, Fourth Edition

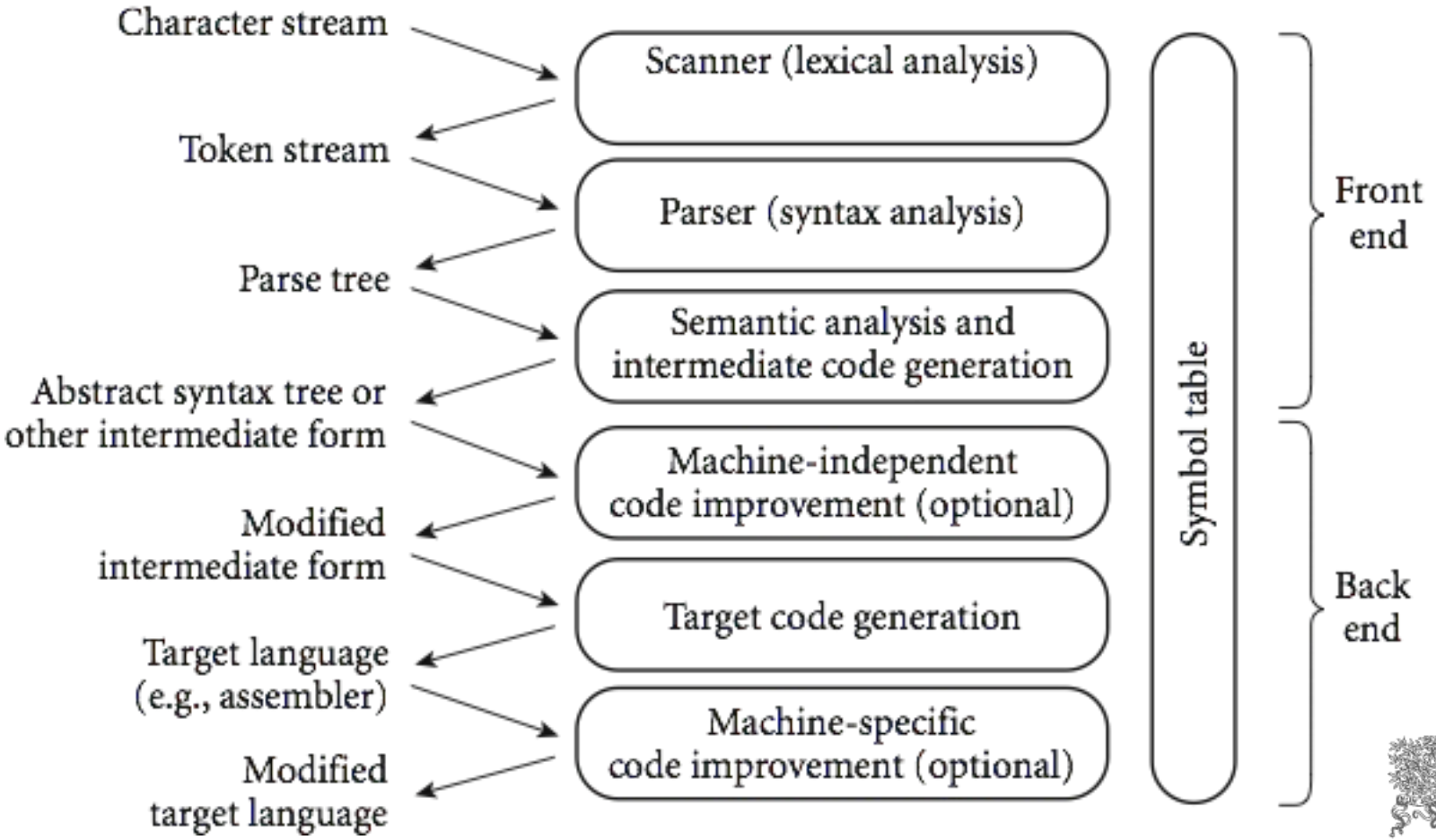
Michael L. Scott



ELSEVIER

An Overview of Compilation

- Phases of Compilation



Semantic Analysis

- Syntax concerns the form of a valid program, while semantics concerns its meaning.
- So far:
  - Lexical Analysis: detects illegal tokens
  - Parsing:
    - detects input with ill-formed parse trees
- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis (last front-end phase)
  - (intermediate) code generation

Why Semantic Analysis ?

- Parsing can not catch some errors
  - Some language constructs are not context-free
  - CFG is not expressive enough
- It allows us to enforce rules (e.g., type consistency)
- It provides the information needed to generate an equivalent output program.
- Consider it the last line of defense for detecting errors

Role of Semantic Analysis

- Semantic Analyzer does checks like:
  - All identifiers are declared
  - Check types
  - Reserved identifiers are not misused
  - Fill in the Symbol Table, and Others...
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a syntax tree (usually first)
  - information gathered is needed by the code generator

Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree
- The notes below discuss attribute grammars and their ad-hoc cousins, ACTION ROUTINES

Attribute Grammars

- We'll start with decoration of parse trees, then consider syntax trees
- Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

Attribute Grammars

$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow E - T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow T / F \\
T &\rightarrow F \\
F &\rightarrow - F
\end{aligned}
$$

- This says nothing about what the program MEANS

Attribute Grammars

- We can turn this into an attribute grammar as follows (similar to Figure 4.1):

```
E → E + T     E1.val = E2.val + T.val
E → E - T     E1.val = E2.val - T.val
E → T              E.val  = T.val
T → T * F     T1.val = T2.val * F.val
T → T / F     T1.val = T2.val / F.val
T → F              T.val  = F.val
F → - F       F1.val = - F2.val
F → (E)       F.val  = E.val
F → const     F.val  = C.val
```

Attribute Grammars

- The attribute grammar serves to define the semantics of the input program
- Attribute rules are best thought of as definitions, not assignments
- They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side

Evaluating Attributes

- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree [see Figure 4.2 for (1+3)*2]
  - When a parse tree under this grammar is fully decorated, the value of the expression will be in the *val* attribute of the root
- The code fragments for the rules are called SEMANTIC FUNCTIONS
  - Strictly speaking, they should be cast as functions, e.g., E1.val = sum (E2.val, T.val), cf., Figure 4.1

Evaluating Attributes

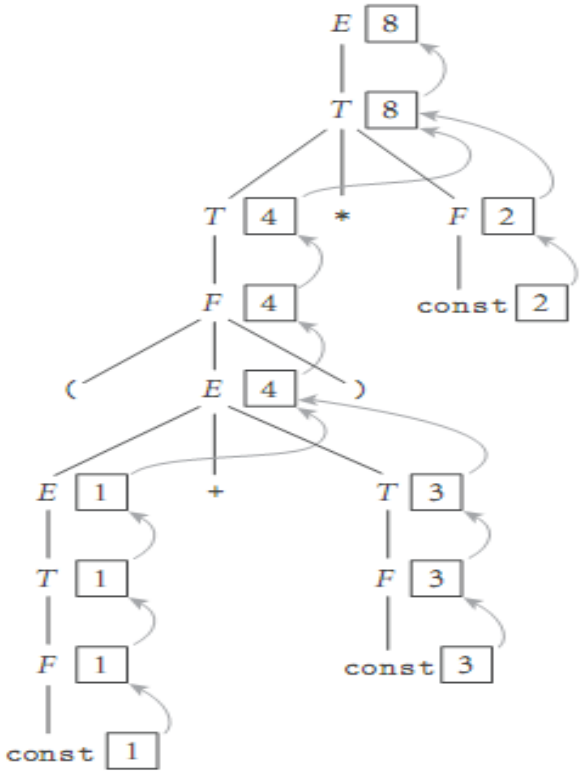


Figure 4.2 Decoration of a parse tree for (1 + 3) * 2, using the attribute grammar of Figure 4.1. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering the box indicate the input(s) to the rule. At the second level of the tree, for example, the two arrows pointing into the box with the 8 represent application of the rule T_1.val := product(T_2.val, F.val).

Evaluating Attributes

- This is a very simple attribute grammar:
  - Each symbol has at most one attribute
    - the punctuation marks have no attributes
- These attributes are all so-called SYNTHESIZED attributes:
  - They are calculated only from the attributes of things below them in the parse tree

Evaluating Attributes

- In general, we are allowed both synthesized and INHERITED attributes:
  - Inherited attributes may depend on things above or to the side of them in the parse tree
  - Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
  - Inherited attributes of the start symbol constitute run-time parameters of the compiler

Evaluating Attributes

- The grammar above is called S-ATTRIBUTED because it uses only synthesized attributes
- Its ATTRIBUTE FLOW (attribute dependence graph) is purely bottom-up
  - It is SLR(1), but not LL(1)
- An equivalent LL(1) grammar requires inherited attributes:

Evaluating Attributes – Example

- Attribute grammar in Figure 4.3:

1. $E \longrightarrow T\ TT$
   - $\triangleright$ TT.st := T.val
   - $\triangleright$ E.val := TT.val
2. $TT_1 \longrightarrow +\ T\ TT_2$
   - $\triangleright$ TT_2.st := TT_1.st + T.val
   - $\triangleright$ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow -\ T\ TT_2$
   - $\triangleright$ TT_2.st := TT_1.st − T.val
   - $\triangleright$ TT_1.val := TT_2.val
4. $TT \longrightarrow \epsilon$
   - $\triangleright$ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   - $\triangleright$ FT.st := F.val
   - $\triangleright$ T.val := FT.val

Evaluating Attributes– Example

- Attribute grammar in Figure 4.3 (continued):

6. $FT_1 \longrightarrow$ `*` $F\ FT_2$
   - ▷ FT_2.st := FT_1.st × F.val
   - ▷ FT_1.val := FT_2.val

7. $FT_1 \longrightarrow$ `/` $F\ FT_2$
   - ▷ FT_2.st := FT_1.st ÷ F.val
   - ▷ FT_1.val := FT_2.val

8. $FT \longrightarrow \epsilon$
   - ▷ FT.val := FT.st

9. $F_1 \longrightarrow$ `-` F_2
   - ▷ F_1.val := − F_2.val

10. $F \longrightarrow$ `(` E `)`
    - ▷ F.val := E.val

11. $F \longrightarrow$ `const`
    - ▷ F.val := const.val

Evaluating Attributes– Example

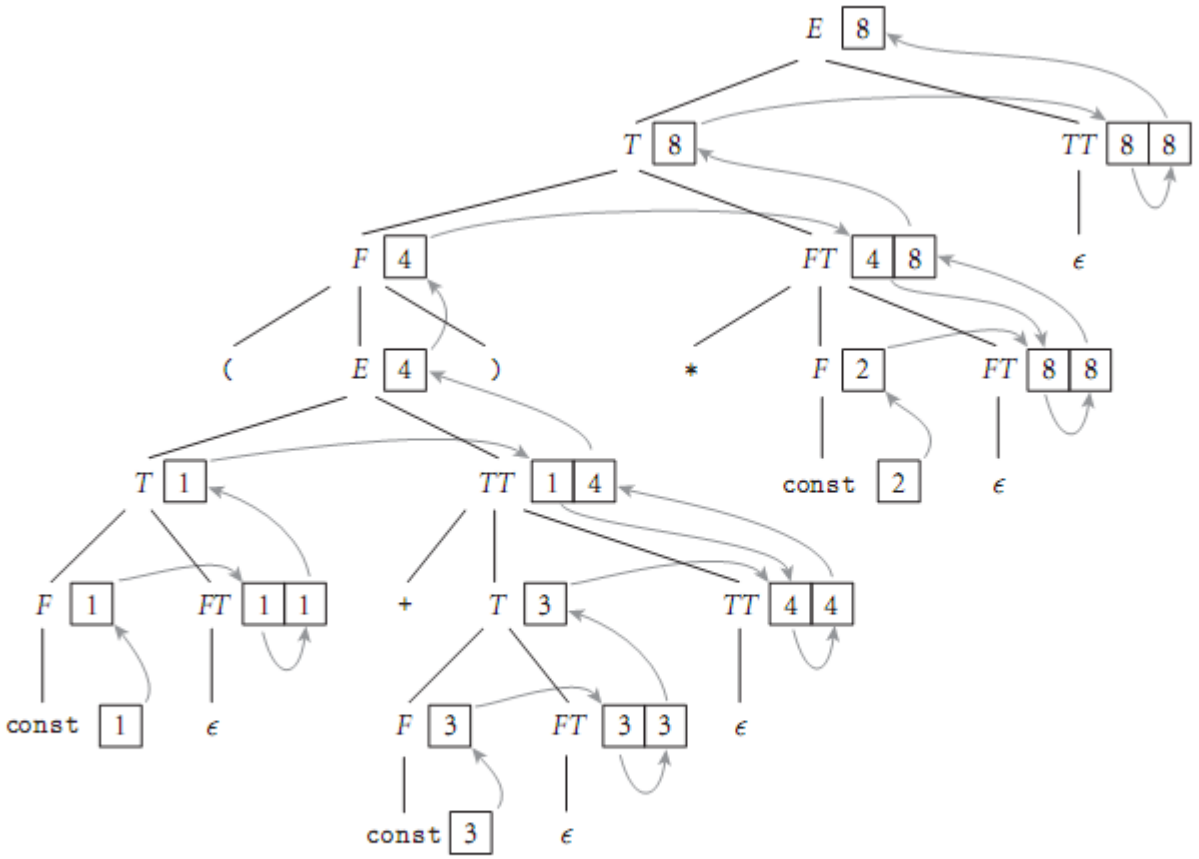


Figure 4.4 Decoration of a top-down parse tree for `(1 + 3) * 2`, using the AG of Figure 4.3. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the st attribute; the right holds `val`.

ELSEVIER

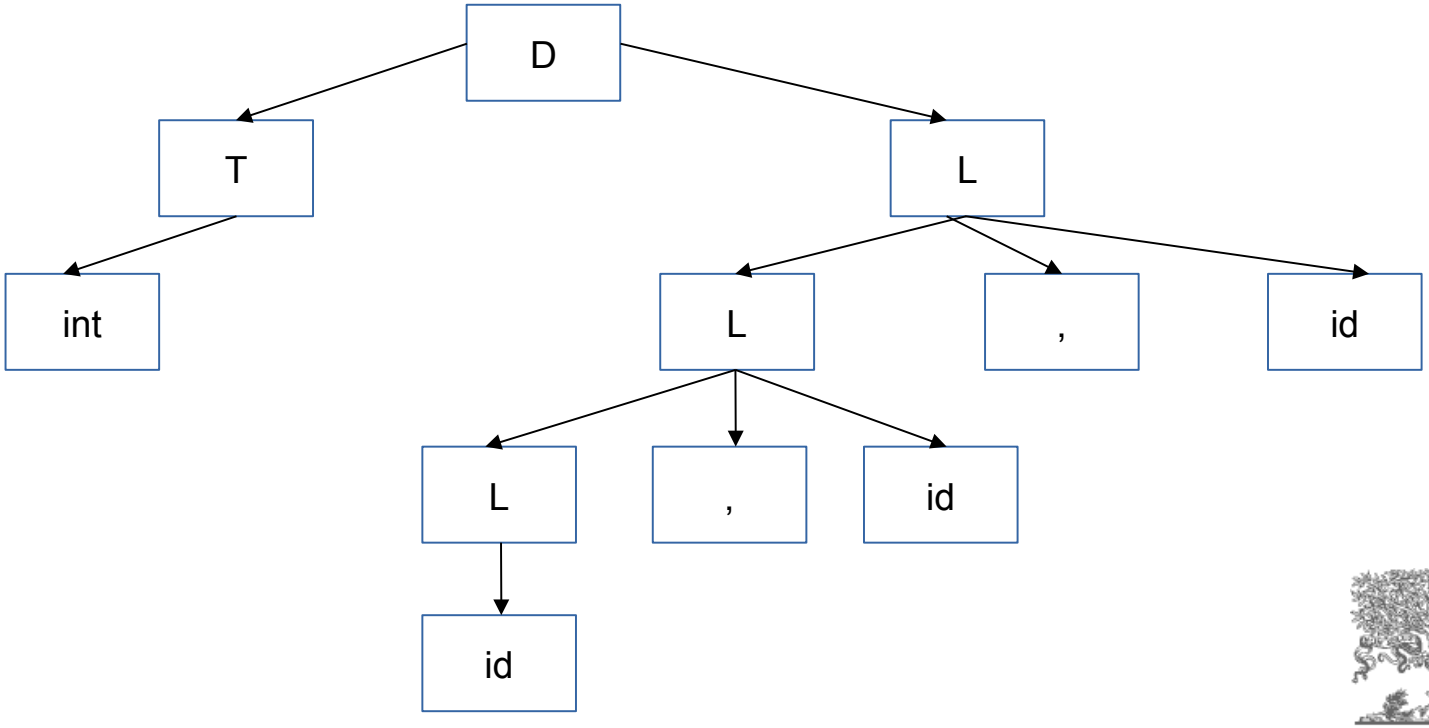
Evaluating Attributes

- Consider this attribute grammar (also called Syntax-Directed Definition), construct the annotated parse tree for the input expression: “int a, b, c”:

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)

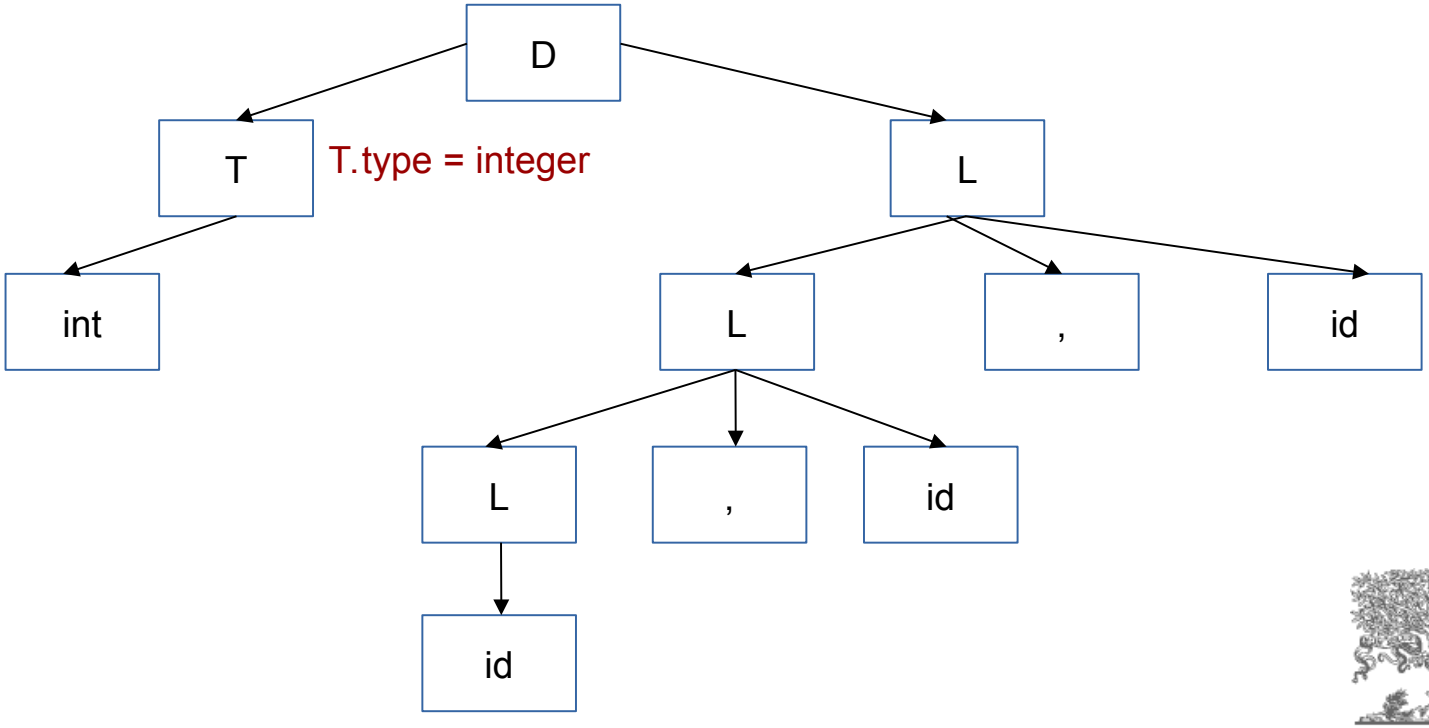
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



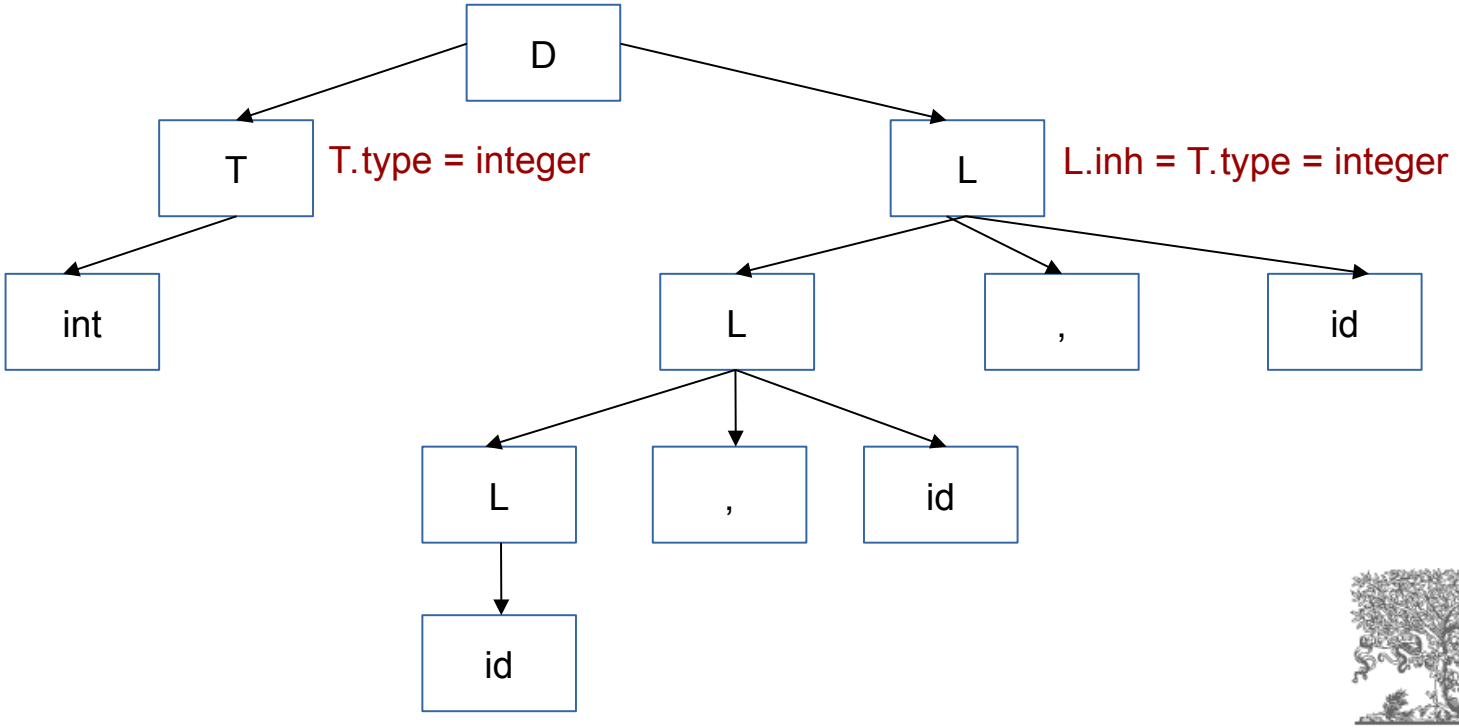
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



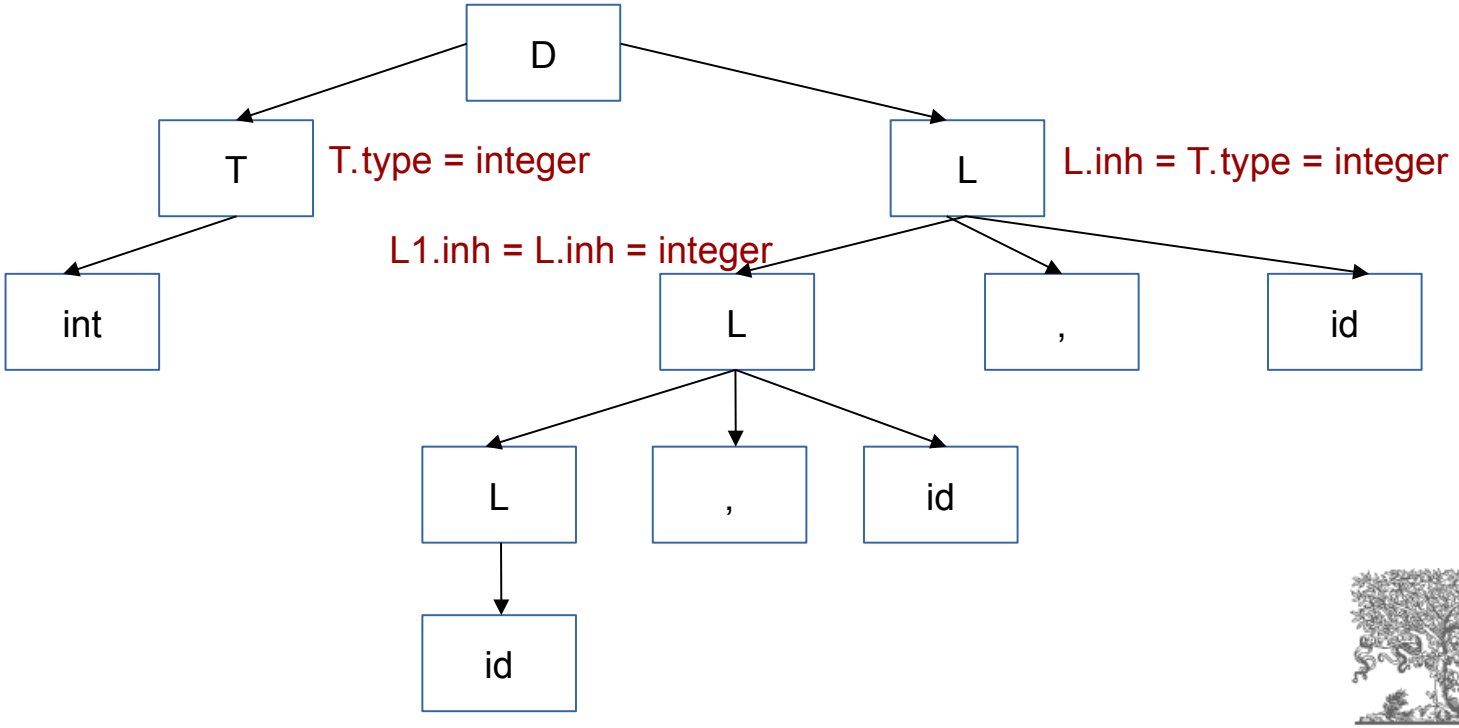
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



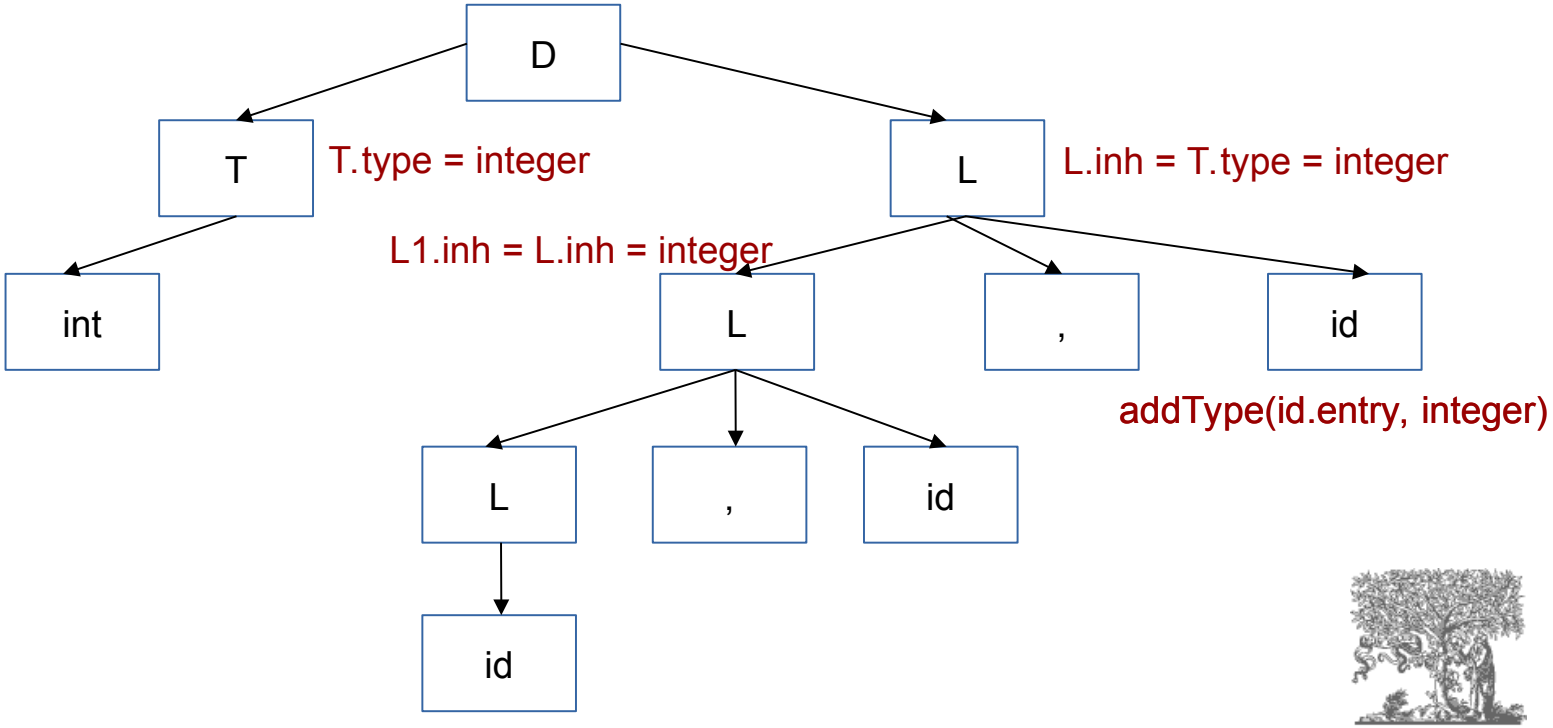
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



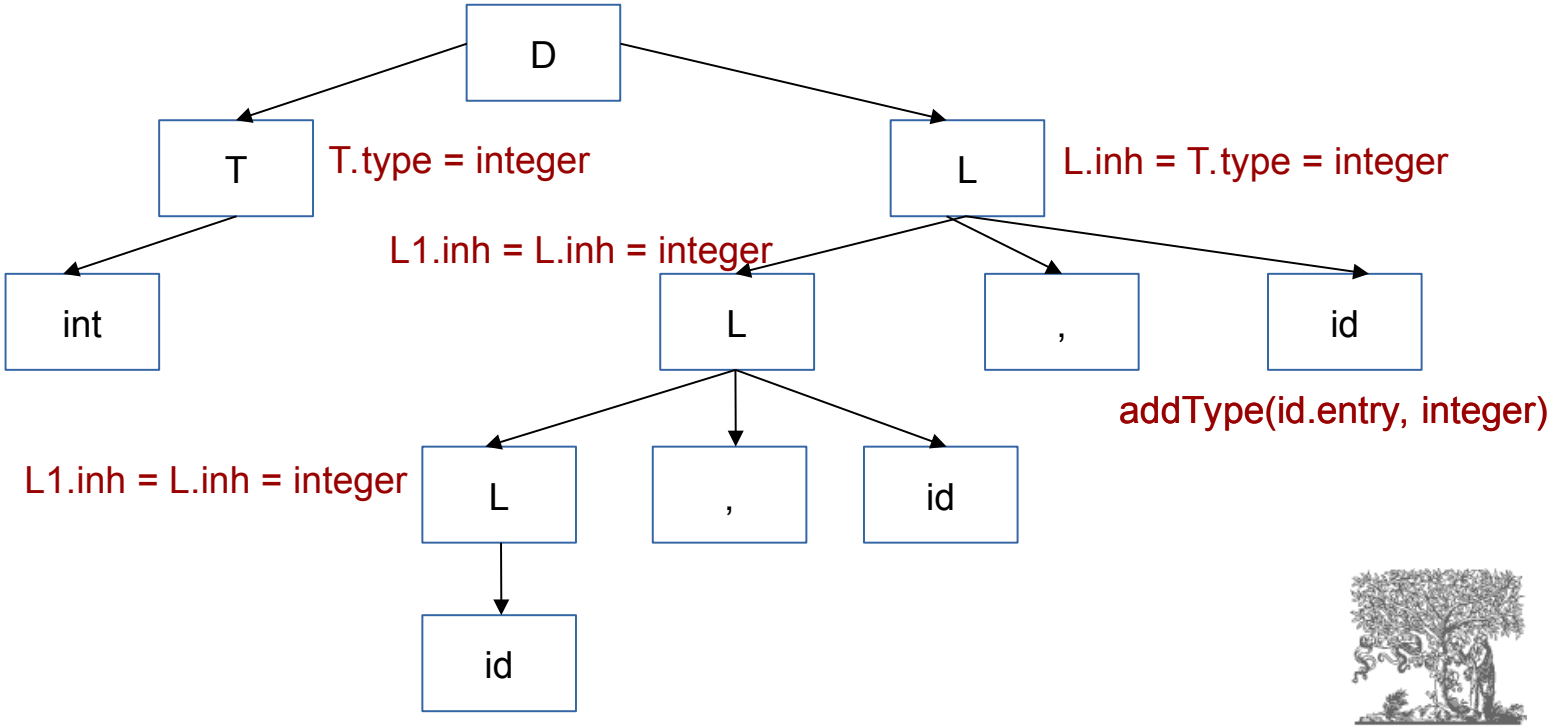
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



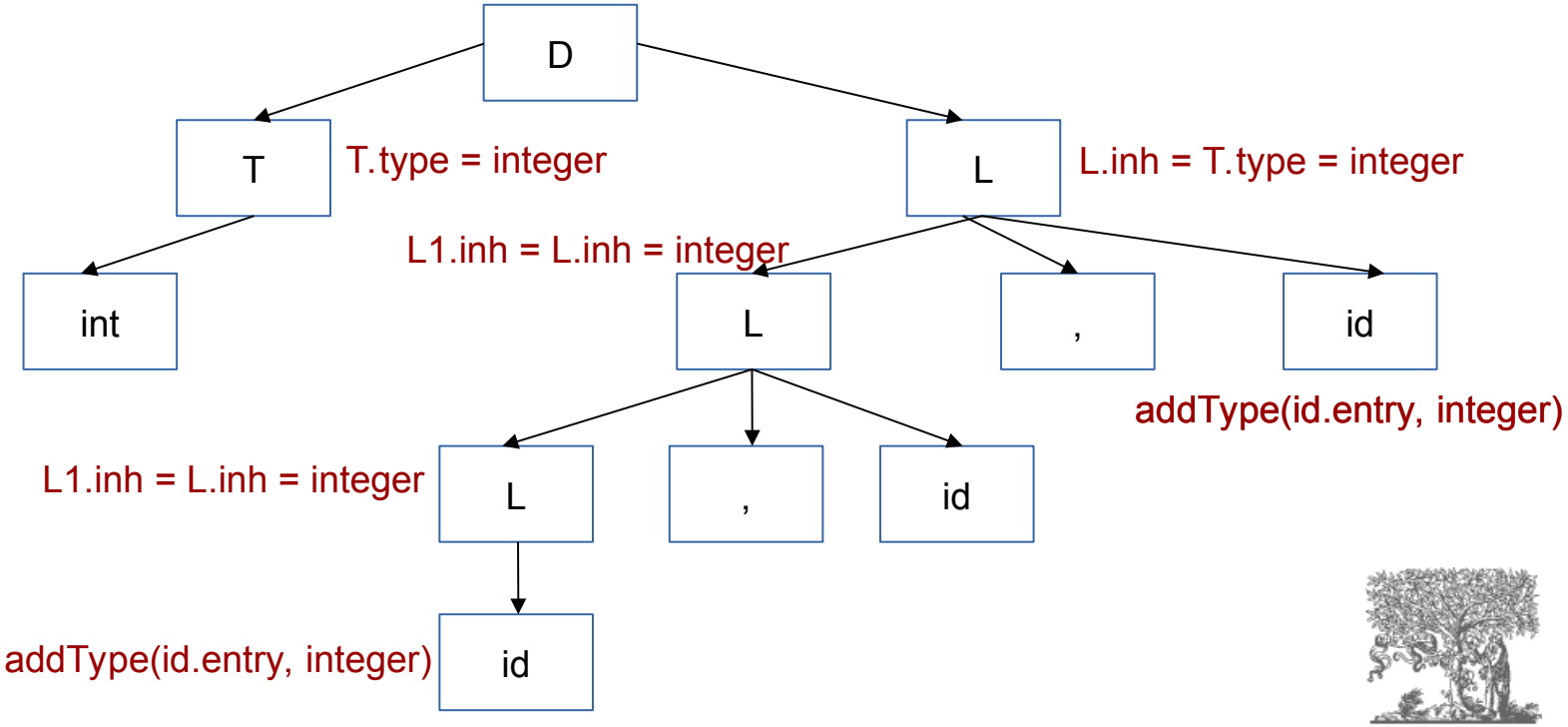
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



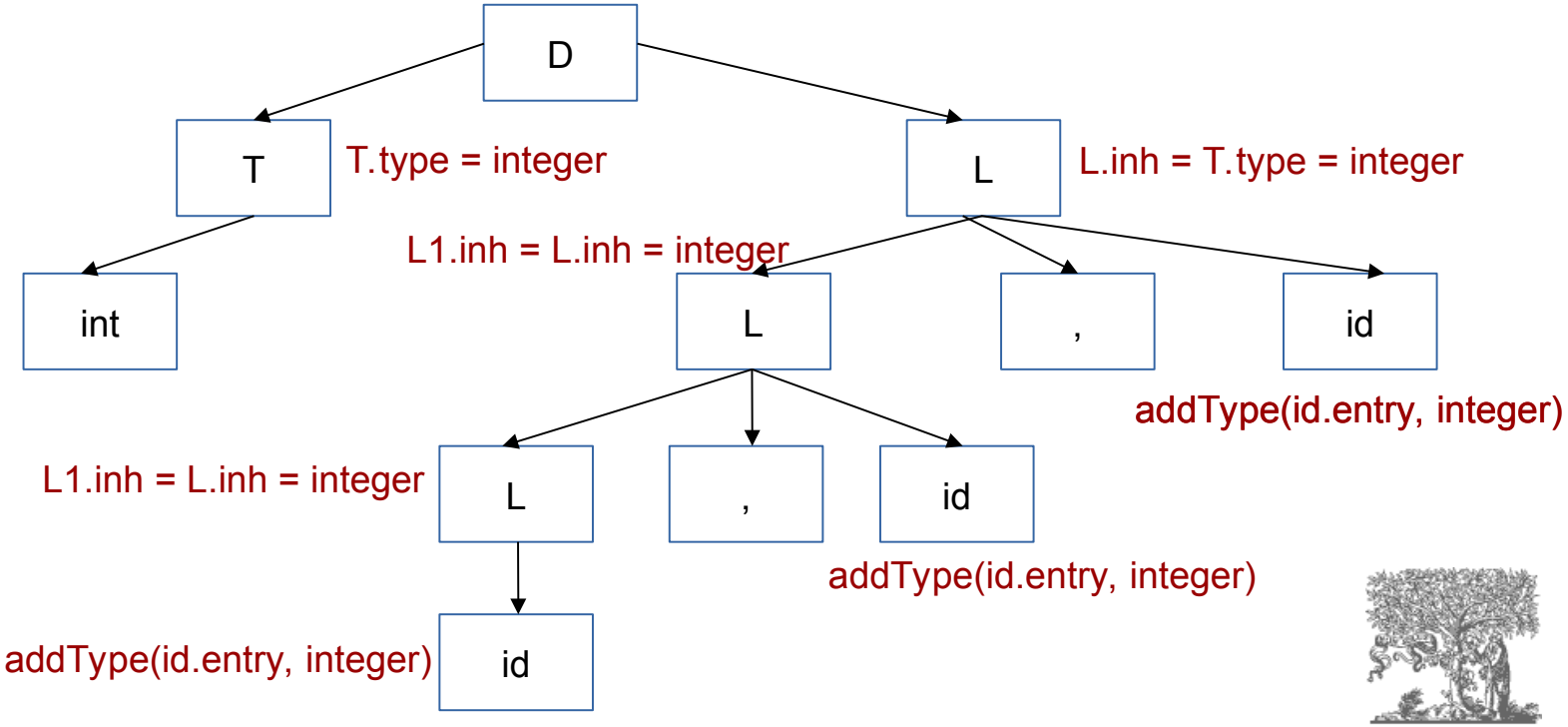
Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh
	addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



Evaluating Attributes

$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.inh = L.inh addType(id.entry,L.inh)
$L \rightarrow$ id	addType(id.entry,L.inh)



Evaluating Attributes – Flow

- LR grammars that are S-attributed can have their attributes computed during bottom-up LR parsing.
- Similarly, LL grammars that are L-attributed can have their attributes computed during a single-pass, top-down LL-parse.

Evaluating Attributes – Flow

- In general attribute grammars do not require that information flows in any particular way. (Although cycles should be avoided.)
- But we can consider attribute grammars in which information only flows in ways that they can be incorporated into a parser.

S- and L- attributed – Eval. main diff.

- S-attributed:
  - Use only synthetic attributes
  - Attributes are evaluated during bottom-up parsing
  - Convenient to evaluate synthesized attributes during an LR parse
- L-attributed:
  - Use synthetic attributes
  - Use Inherited attributes from the left, and hence the name L-attributed. If the production is A → X1X2...Xn, then the inherited attributes for Xj can depend only on:
    - Inherited attributes of A (LHS)
    - Any attribute of X1, ..., Xj-1, i.e. left of Xj.

Syntax Tree

- A parse tree, also known as a **concrete syntax tree**, shows how a program can be derived from its grammar.
- The parse tree will reflect the program's abstract structure.
- Syntax tree, also known as **abstract syntax tree**, solves some problems of the parse tree.
- Again why syntax tree?

Syntax Tree

- Recall the grammar: E → int | (E) | E + E

  Input: **3 + (1 + 6)**

Syntax Tree

- Recall the grammar: E → int | (E) | E + E

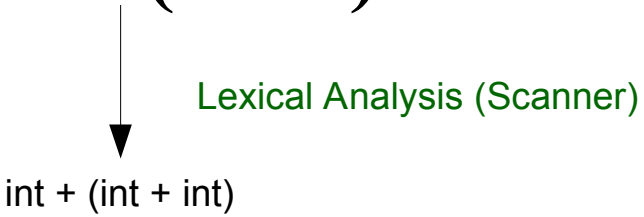
Input: **3 + (1 + 6)**

Lexical Analysis (Scanner)

Syntax Tree

- Recall the grammar: E → int | (E) | E + E

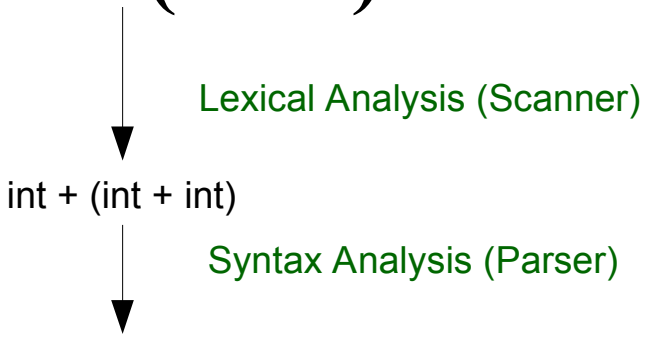
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

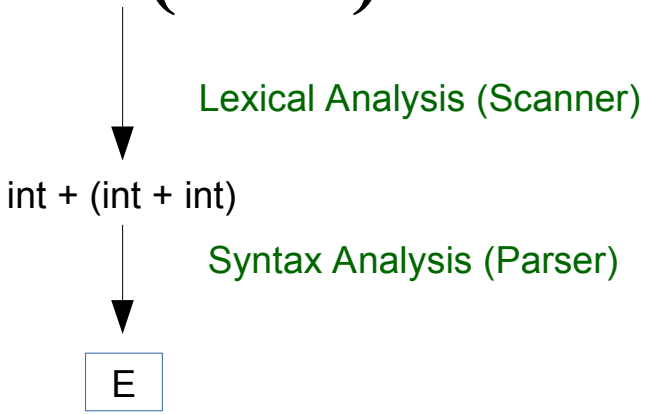
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

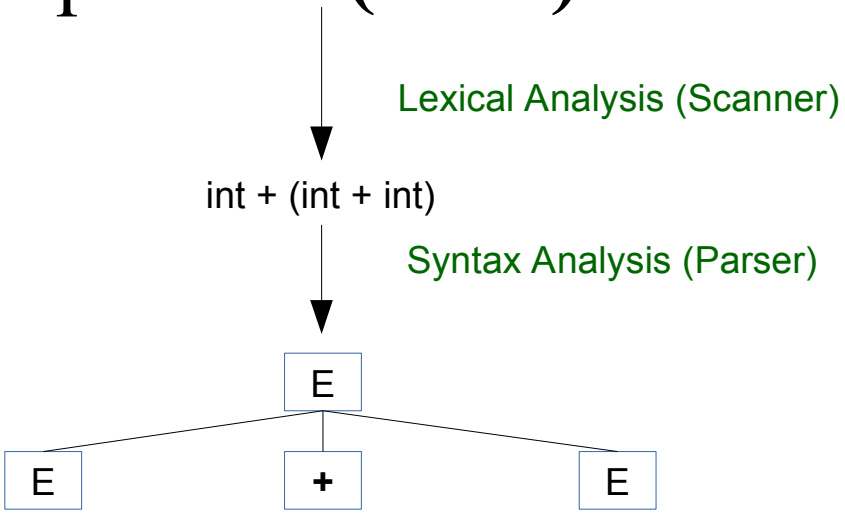
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

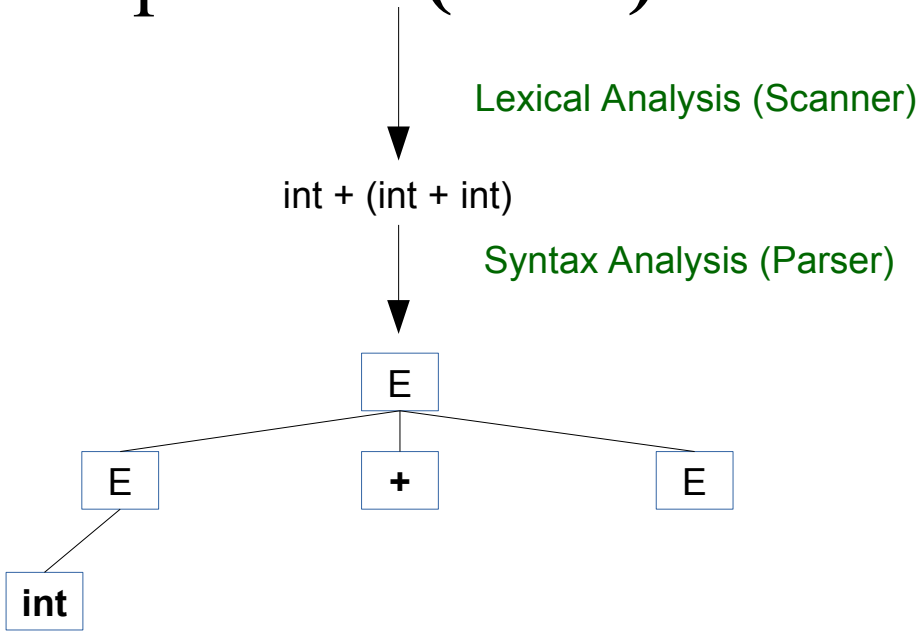
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

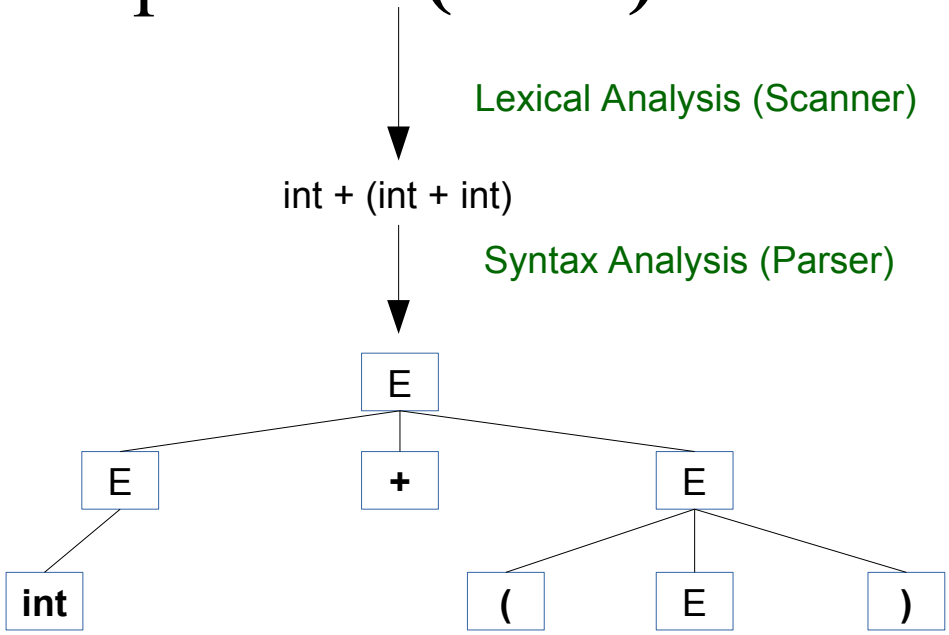
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

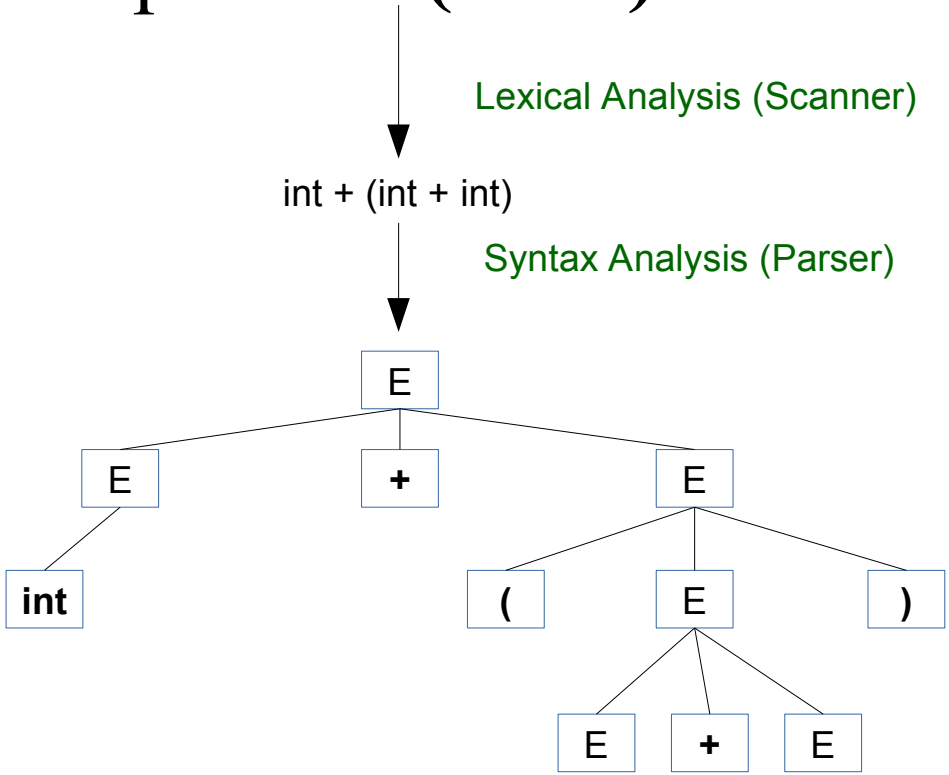
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

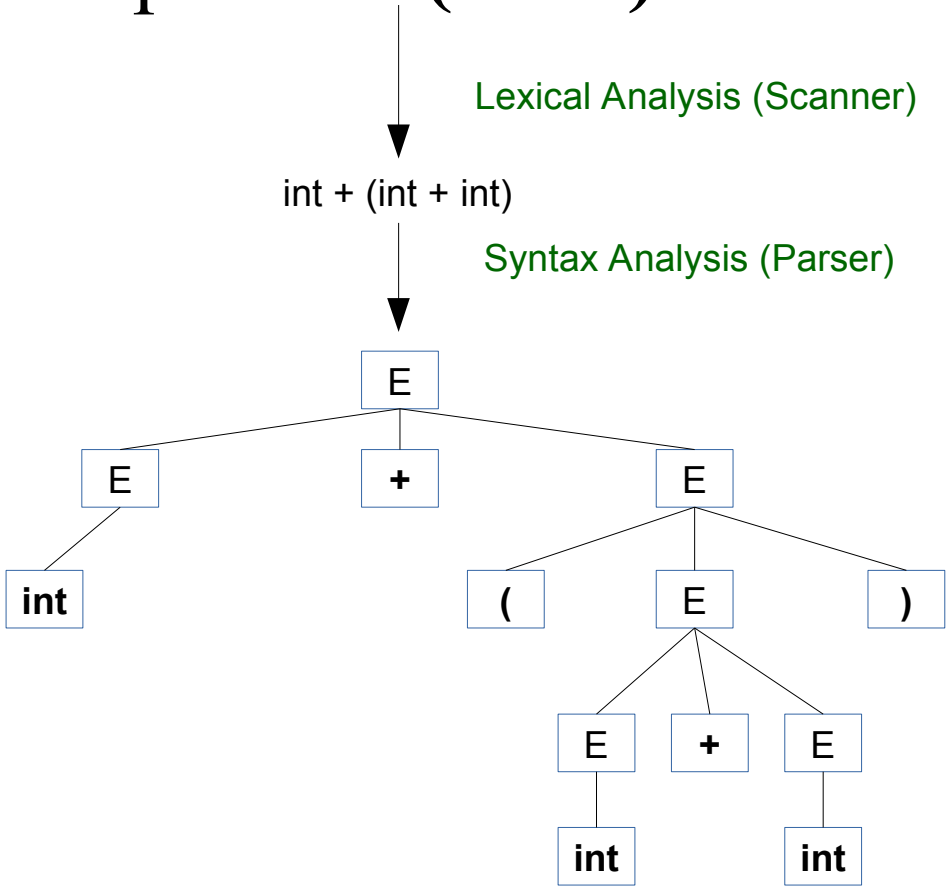
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

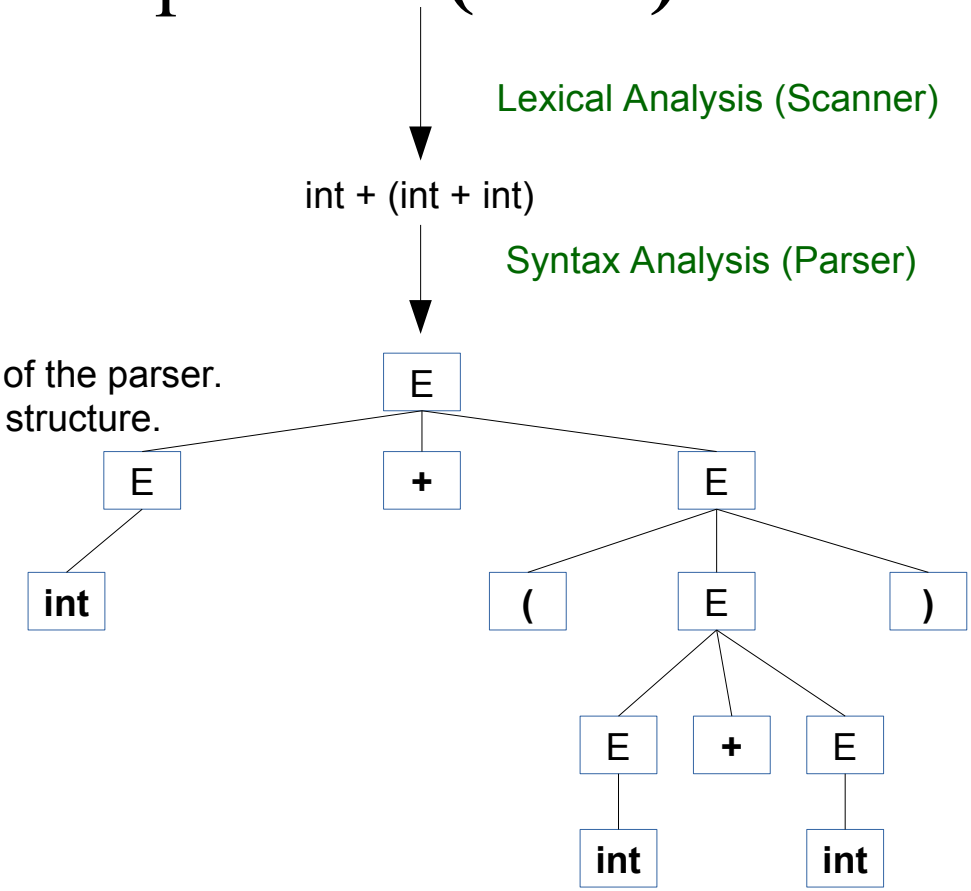
Input: **3 + (1 + 6)**



Syntax Tree

- Recall the grammar: E → int | (E) | E + E

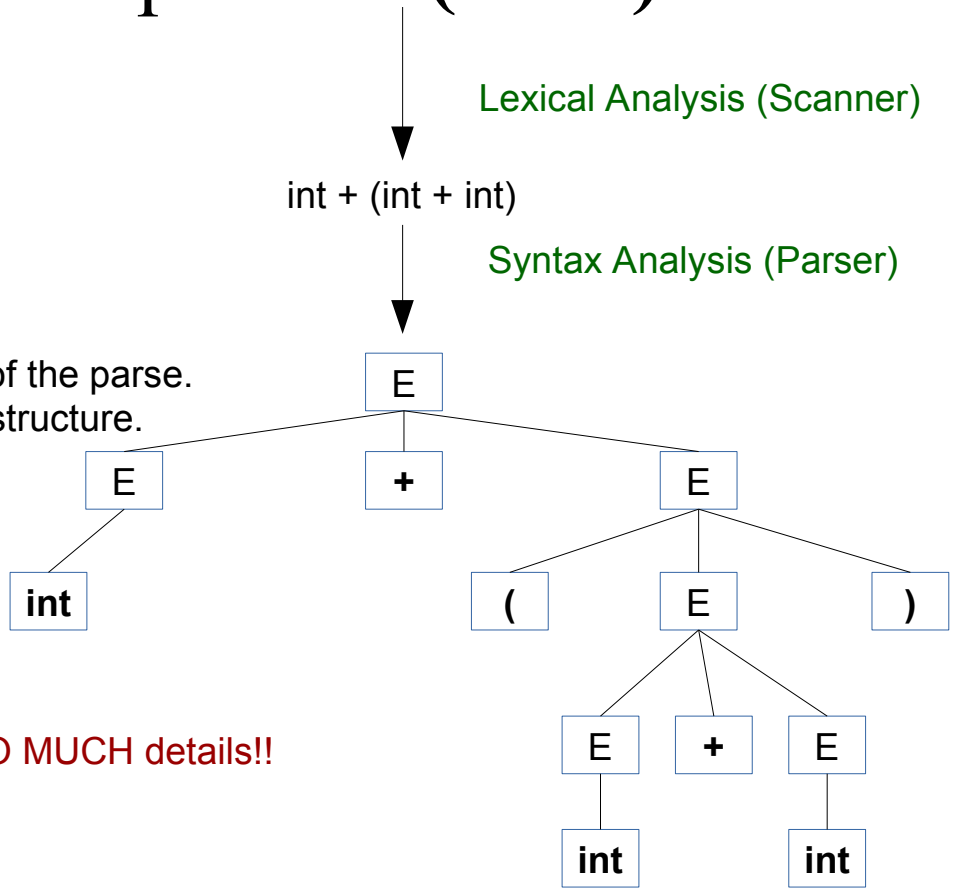
Input: **3 + (1 + 6)**



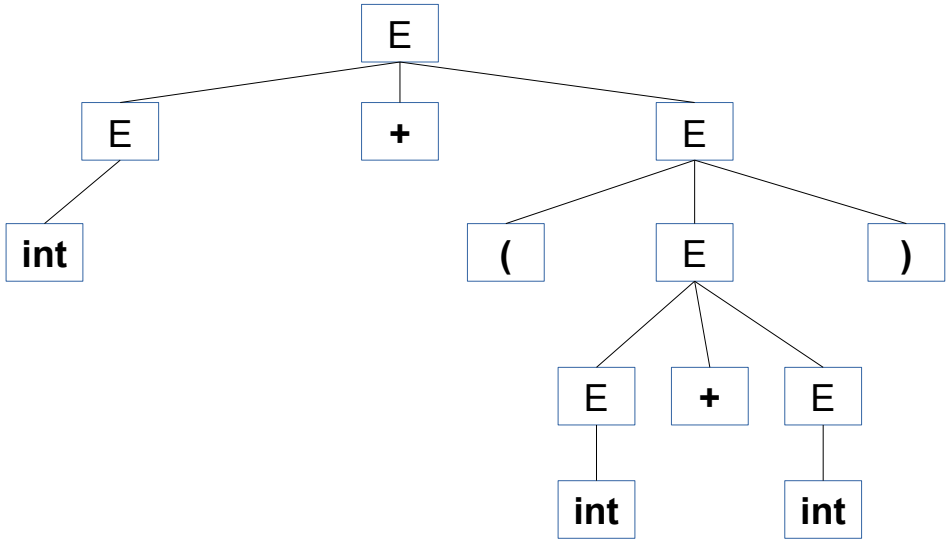
Syntax Tree

- Recall the grammar: E → int | (E) | E + E

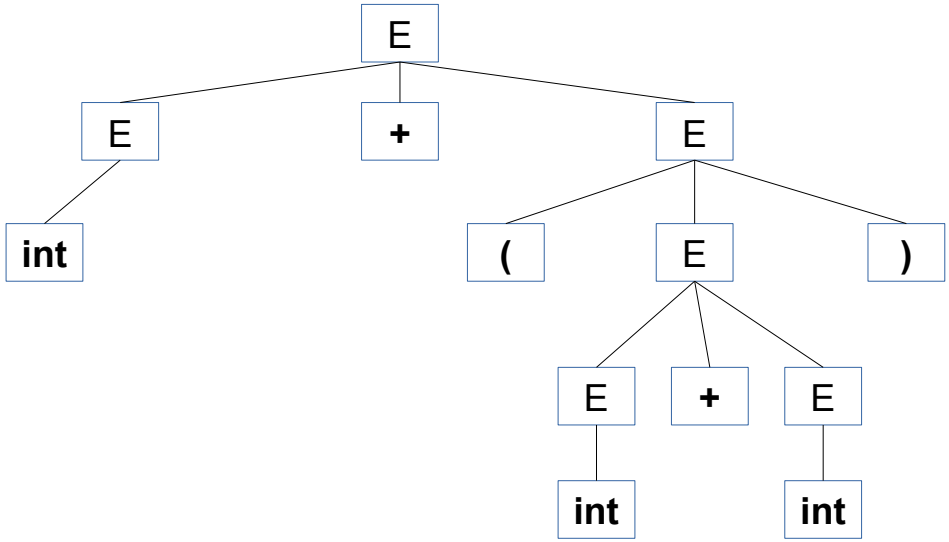
Input: **3 + (1 + 6)**



Syntax Tree

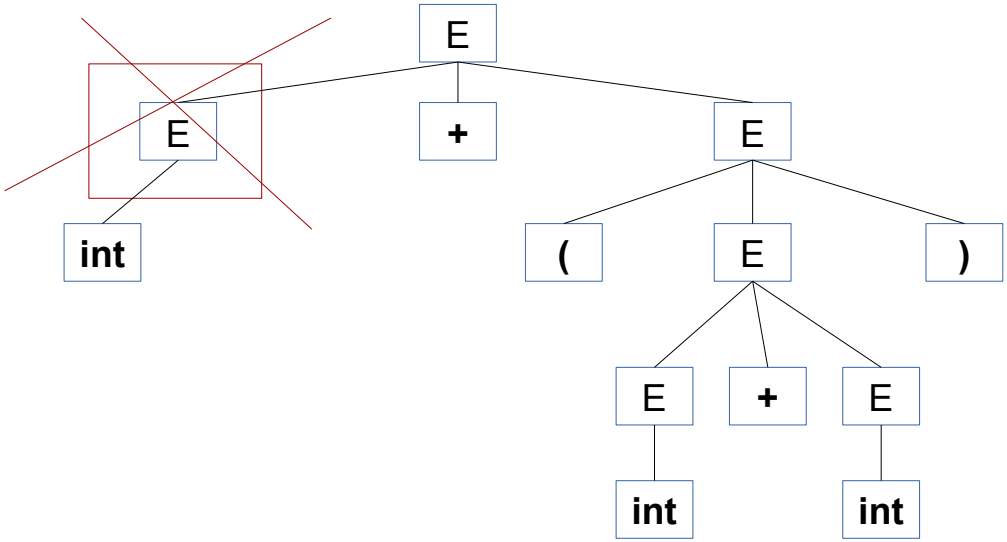


Syntax Tree



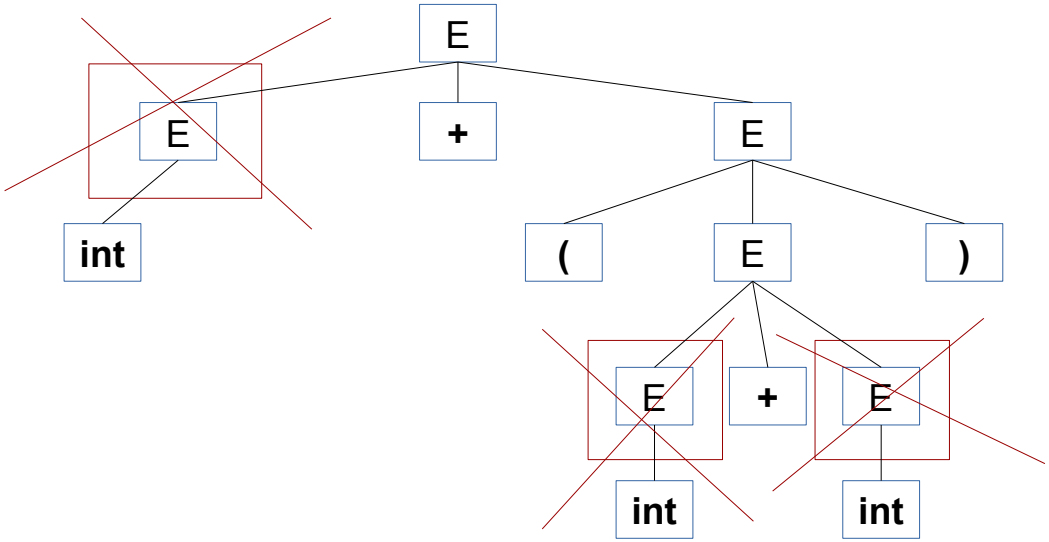
- Node ‘E’ has only child, do we really need it?

Syntax Tree



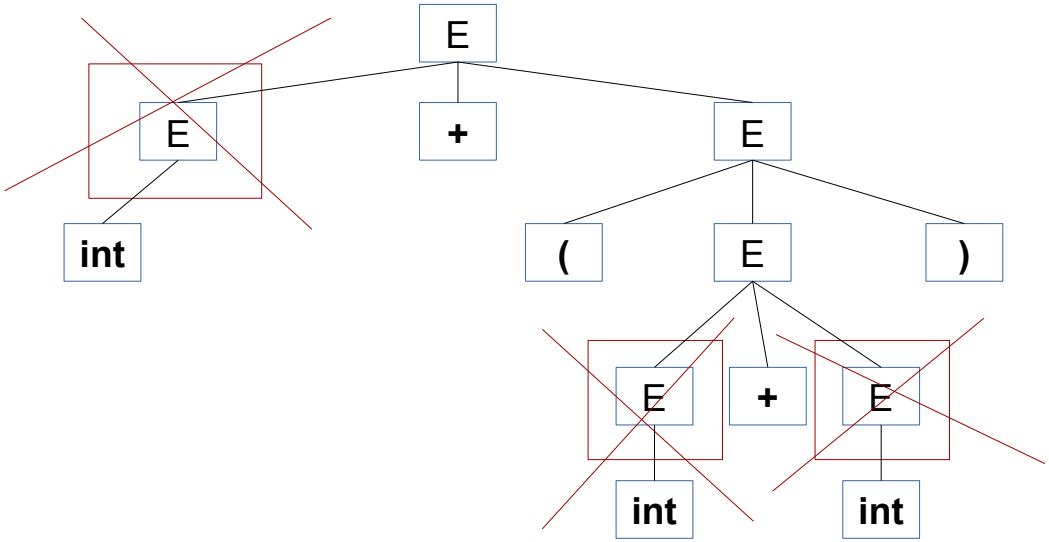
- Node ‘E’ has only child, do we really need it?

Syntax Tree



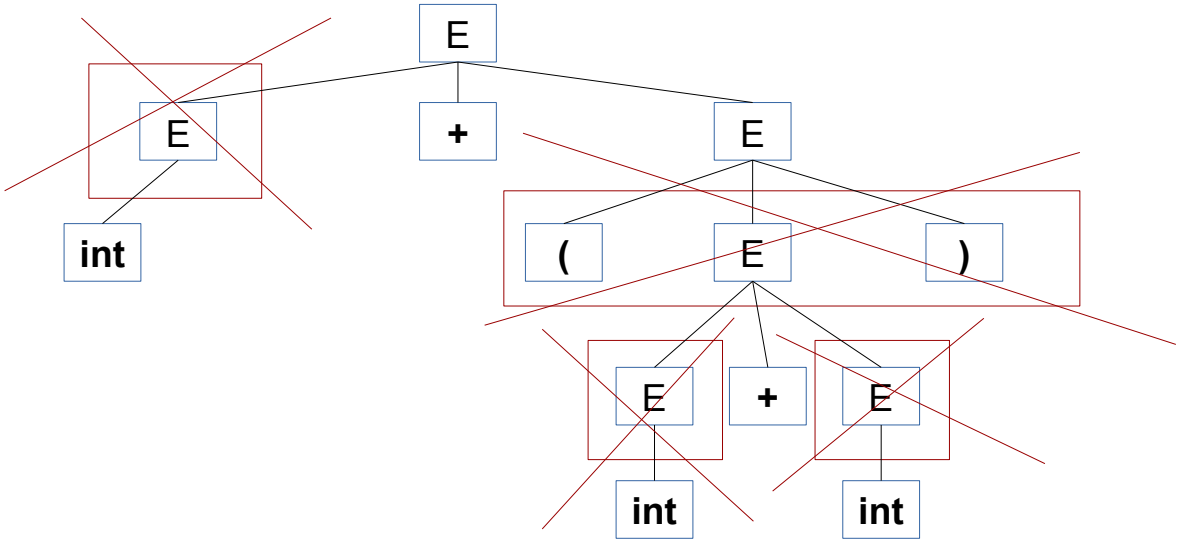
- Node ‘E’ has only child, do we really need it?

Syntax Tree



- Node ‘E’ has only child, do we really need it?
- ‘(E)’ is important since it capture the association.
  - But, the tree structure already capture that, do we really need it then?

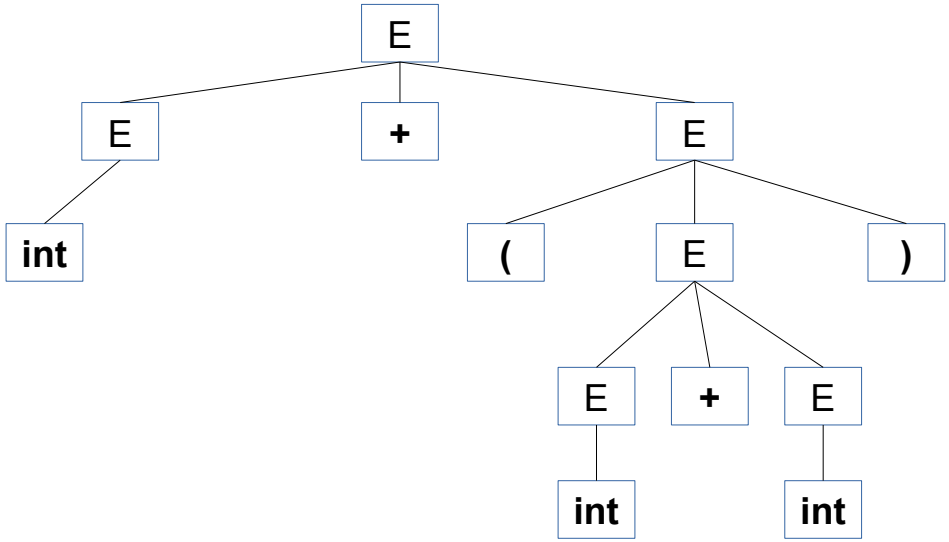
Syntax Tree



- Node ‘E’ has only child, do we really need it?
- ‘(E)’ is important since it capture the association.
  - But, the tree structure already capture that, do we really need it then?

Syntax Tree

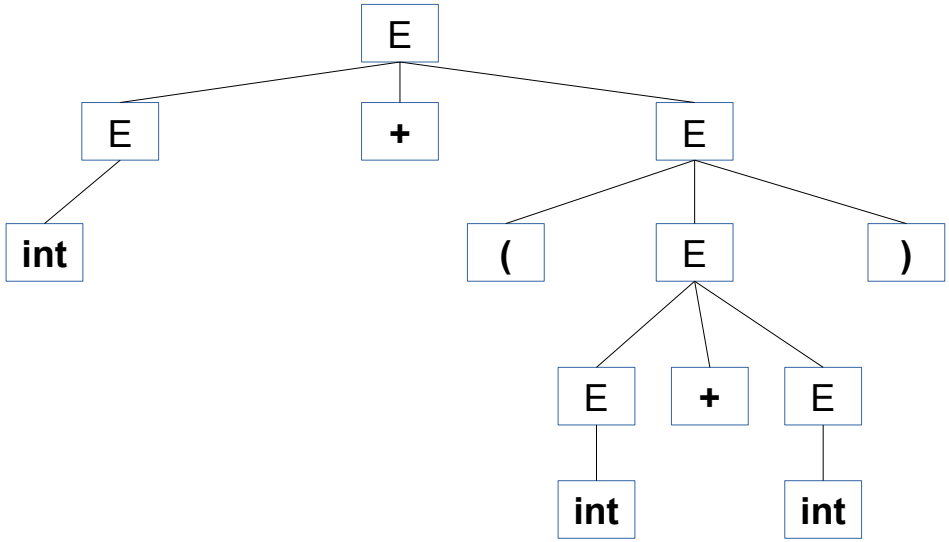
In: int + (int + int)



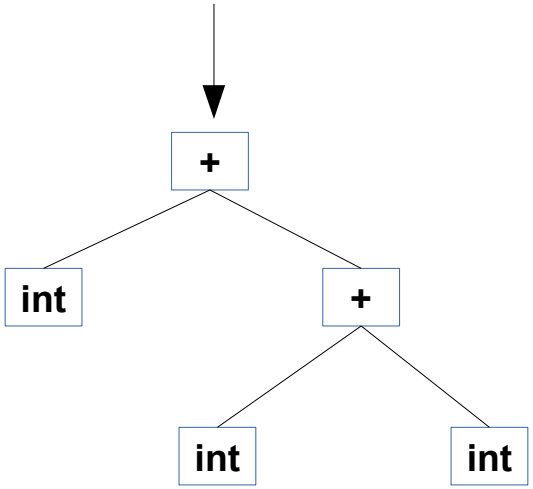
Parse Tree

Syntax Tree

In: int + (int + int)



Parse Tree



Abstract Syntax Tree

Syntax Tree

- Syntax tree for a simple program to print an average of an integer and a real

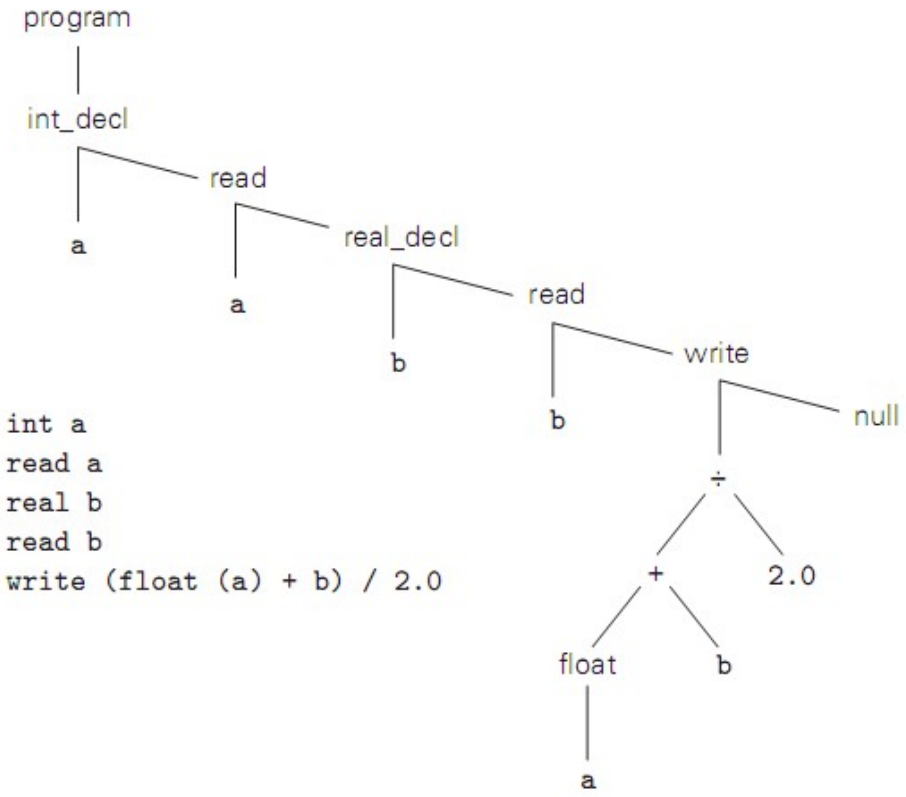


Figure 4.12 Syntax tree for a simple calculator program.

Syntax Tree

- Condensed form of a parse tree.
- It is useful as:
  - A notation for describing programs on paper (including when doing formal analysis)
  - A compiler intermediate representation
- It is easier to convert to target code.