Semantic Analysis

CSCI 3136

Principles of Programming Languages

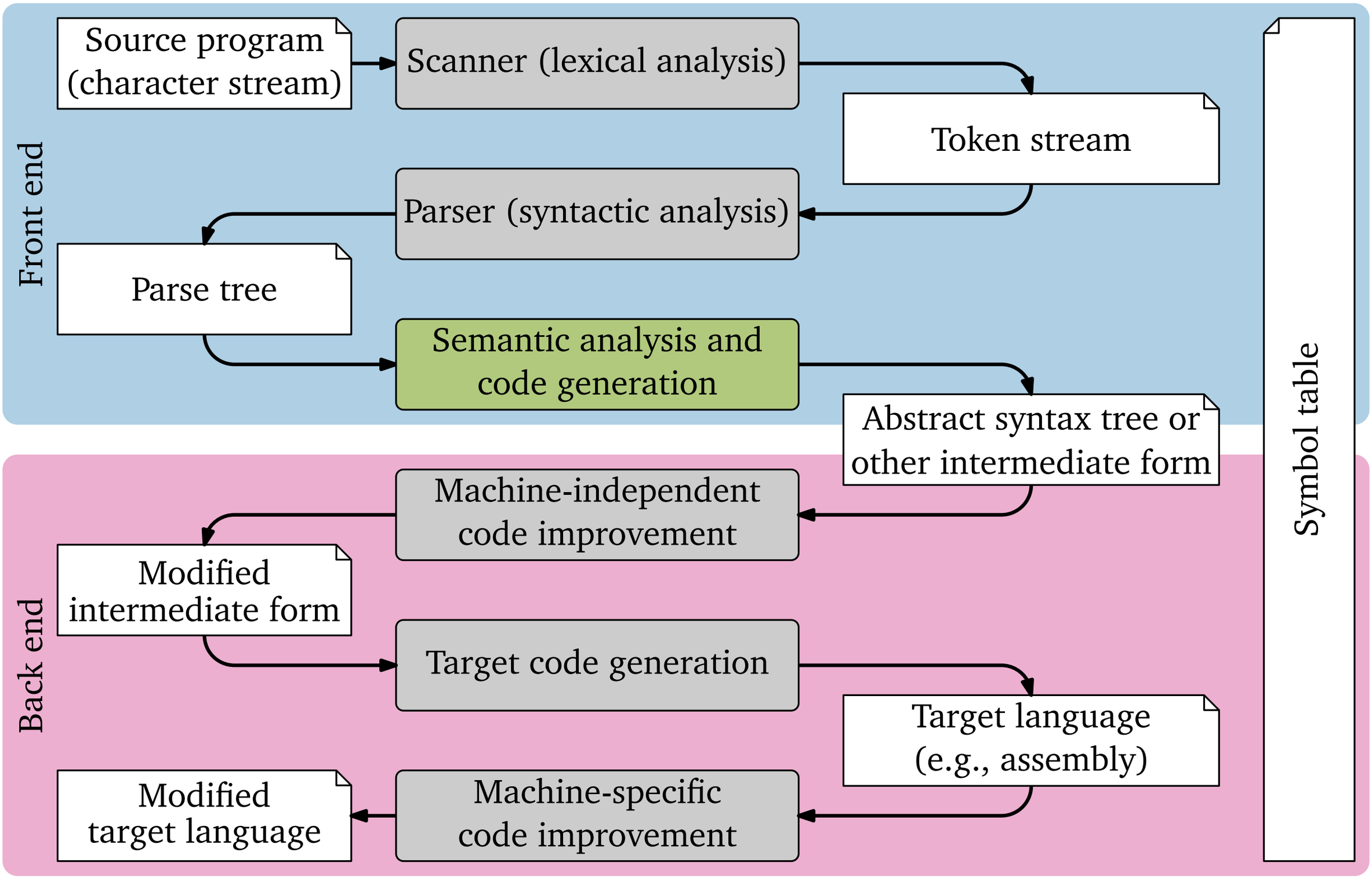
Faculty of Computer Science

Dalhousie University

Winter 2012

Reading: Chapter 4

Motivation


Front end
Source program (character stream)
Scanner (lexical analysis)
Token stream
Parser (syntactic analysis)
Parse tree
Semantic analysis and code generation
Abstract syntax tree or other intermediate form
Symbol table
Back end
Machine-independent code improvement
Modified intermediate form
Target code generation
Target language (e.g., assembly)
Modified target language
Machine-specific code improvement

Syntax and Semantics

Syntax

- Describes form of a valid program
- Can be described by a context-free grammar

Semantics

- Describes meaning of a program
- Cannot be be described by a context-free grammar

Some constraints that may appear syntactic are enforced by semantic analysis:

- E.g., use of identifier only after its declaration

Semantic Analysis

- Enforces semantic rules
- Builds intermediate representation (e.g., abstract syntax tree)
- Fills symbol table
- Passes results to intermediate code generator

Two approaches

- Interleaved with syntactic analysis
- As a separate phase

Formal mechanism: Attributes grammars

Enforcing Semantic Rules

Static semantic rules

- Enforced by compiler at compile time
- Example: Do not use undeclared variable

Dynamic semantic rules

- Compiler generates code for enforcement at run time
- Examples: division by zero, array index out of bounds
- Some compilers allow these checks to be disabled

Attribute Grammars

An ***attribute grammar*** is an augmented context-free grammar:

- Each symbol in a production has a number of attributes.
- Each production is augmented with semantic rules that
  - Copy attribute values between symbols,
  - Evaluate attribute values using semantic functions,
  - Enforce constraints on attribute values.

Attribute Grammars: Example

$$E \longrightarrow E + T$$

$$E \longrightarrow E - T$$

$$E \longrightarrow T$$

$$T \longrightarrow T * F$$

$$T \longrightarrow T \ / \ F$$

$$T \longrightarrow F$$

$$F \longrightarrow -F$$

$$F \longrightarrow (\ E \ )$$

$$F \longrightarrow const$$

Attribute Grammars: Example

$E \rightarrow E + T$	$E_1 \rightarrow E_2 + \mathtt{T}$	$\triangleright\ E_1.val := \text{sum}(E_2.val, T.val)$
$E \rightarrow E - T$	$E_1 \rightarrow E_2 - \mathtt{T}$	$\triangleright\ E_1.val := \text{difference}(E_2.val, T.val)$
$E \rightarrow T$	$E \rightarrow T$	$\triangleright\ E.val := T.val$
$T \rightarrow T * F$	$T_1 \rightarrow T_2 * F$	$\triangleright\ T_1.val := \text{product}(T_2.val, F.val)$
$T \rightarrow T\ /\ F$	$T_1 \rightarrow T_2\ /\ F$	$\triangleright\ T_1.val := \text{quotient}(T_2.val, F.val)$
$T \rightarrow F$	$T \rightarrow F$	$\triangleright\ T.val := F.val$
$F \rightarrow -F$	$F_1 \rightarrow -F_2$	$\triangleright\ F_1.val := \text{negate}(F_2.val)$
$F \rightarrow (\ E\ )$	$F \rightarrow (\ E\ )$	$\triangleright\ F.val := E.val$
$F \rightarrow const$	$F \rightarrow const$	$\triangleright\ F.val := const.val$

Synthesized Attributes (Bottom-Up)

The language

$$\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\} = \{\mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}$$

is not context-free but can be recognized by an attribute grammar.

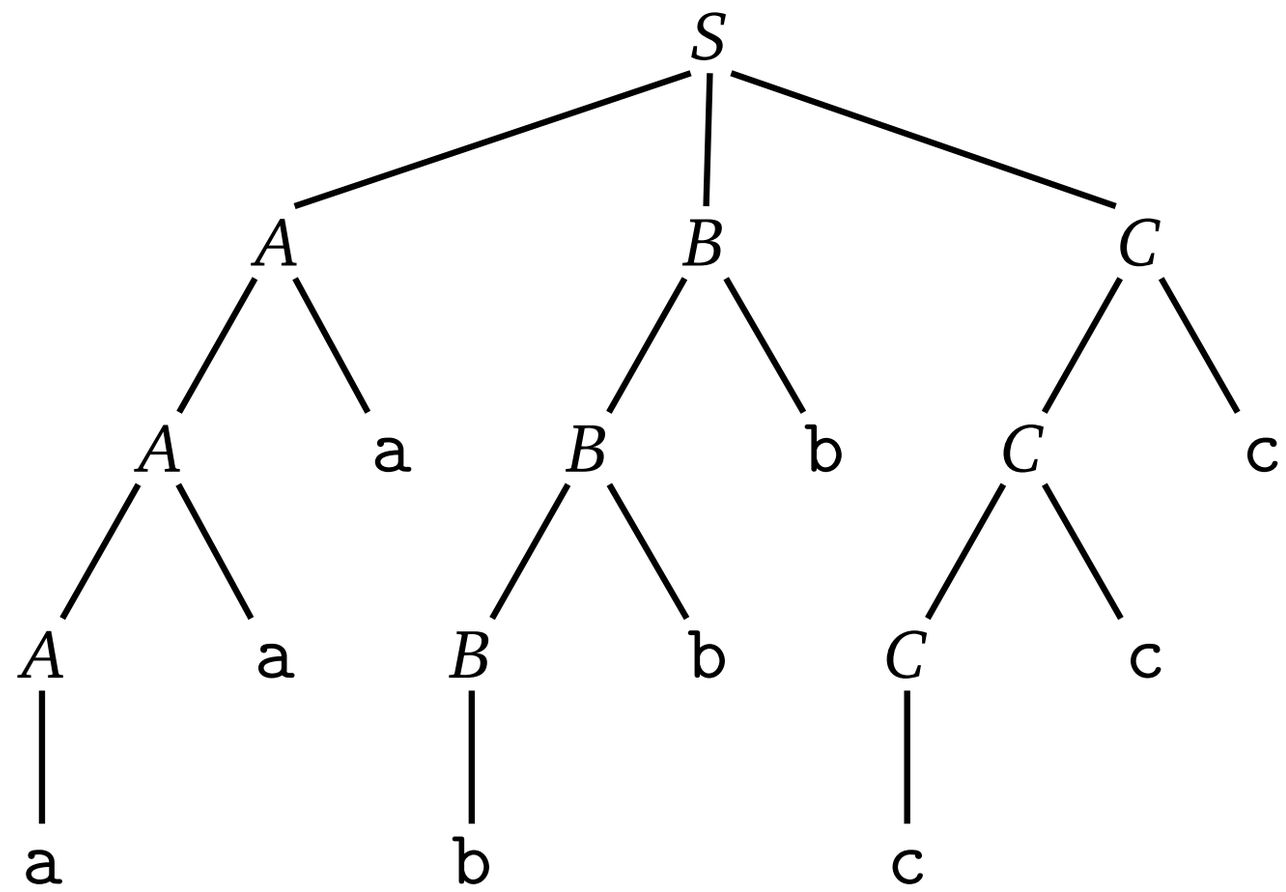
Synthesized attributes: Attributes of LHS are computed from attributes of RHS.

$$
\begin{aligned}
S &\to ABC && \triangleright \text{ Condition: } A.count = B.count = C.count \\
A &\to \mathtt{a} && \triangleright\ A.count := 1 \\
A_1 &\to A_2\ \mathtt{a} && \triangleright\ A_1.count := A_2.count + 1 \\
B &\to \mathtt{b} && \triangleright\ B.count := 1 \\
B_1 &\to B_2\ \mathtt{b} && \triangleright\ B_1.count := B_2.count + 1 \\
C &\to \mathtt{c} && \triangleright\ C.count := 1 \\
C_1 &\to C_2\ \mathtt{c} && \triangleright\ C_1.count := C_2.count + 1
\end{aligned}
$$

Synthesized Attributes: Parse Tree Decoration (1)

Input: `aaabbbccc`

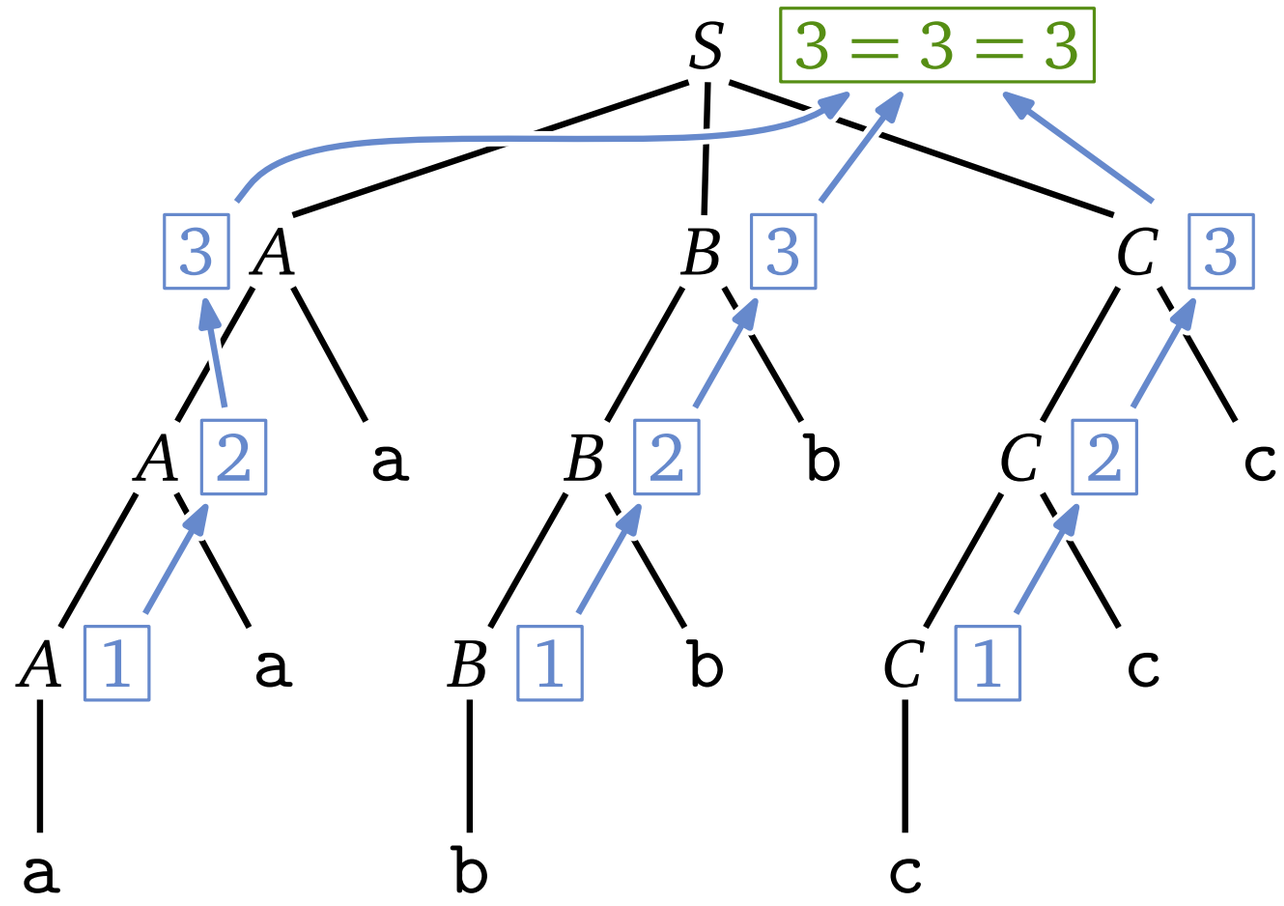
Parse tree



Synthesized Attributes: Parse Tree Decoration (1)

Input: `aaabbbccc`

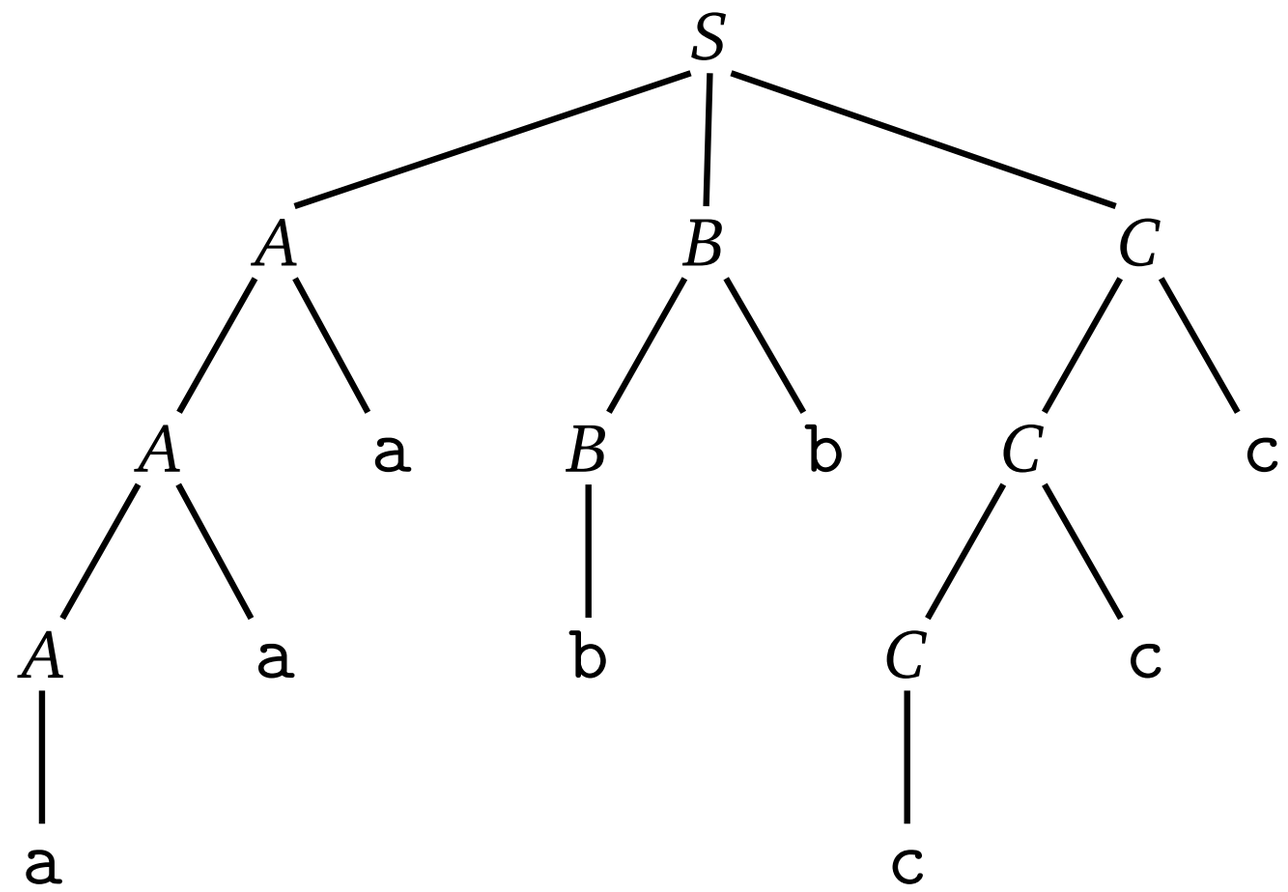
Parse tree



Synthesized Attributes: Parse Tree Decoration (2)

Input: `aaabbccc`

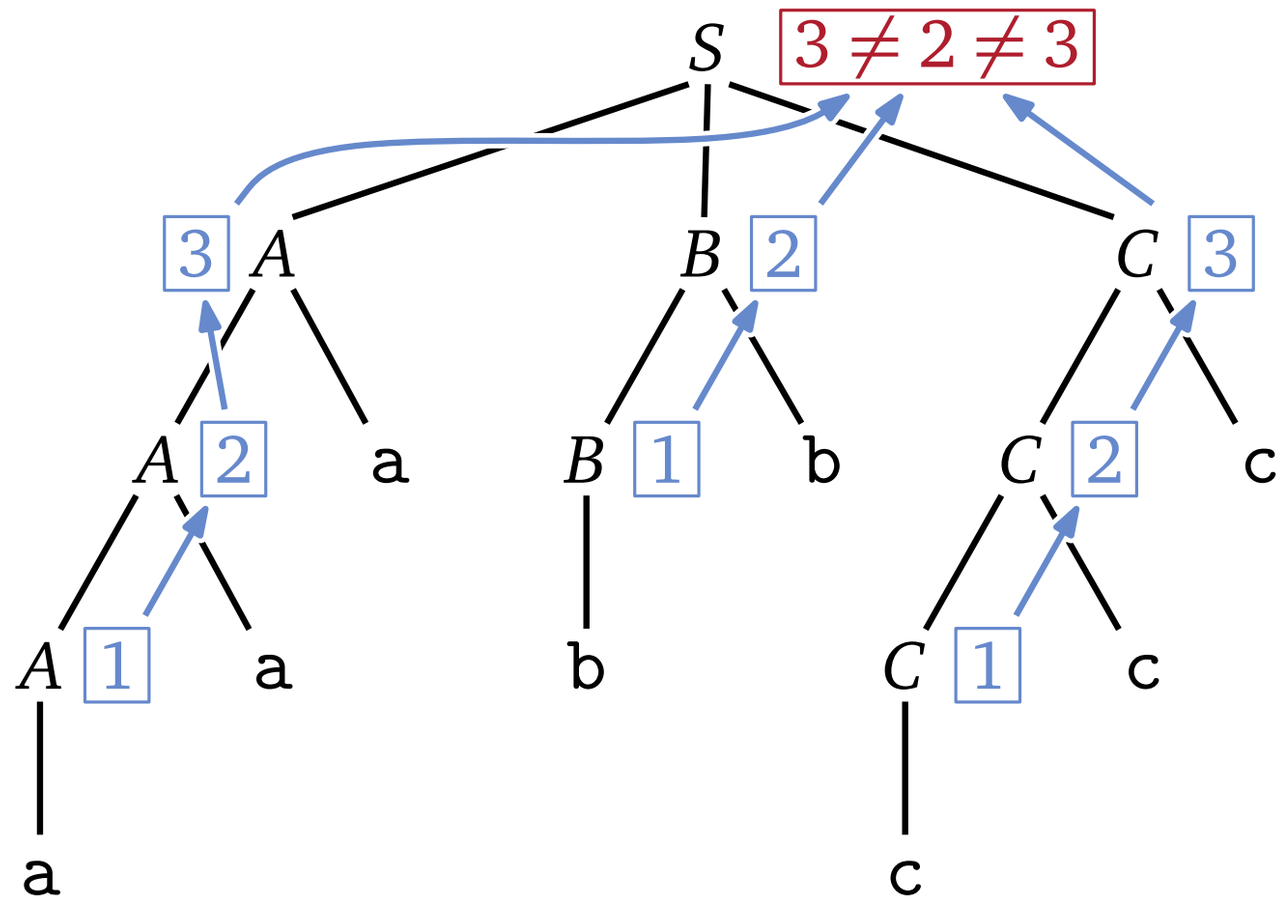
Parse tree



Synthesized Attributes: Parse Tree Decoration (2)

Input: `aaabbccc`

Parse tree



Inherited Attributes (Top-Down)

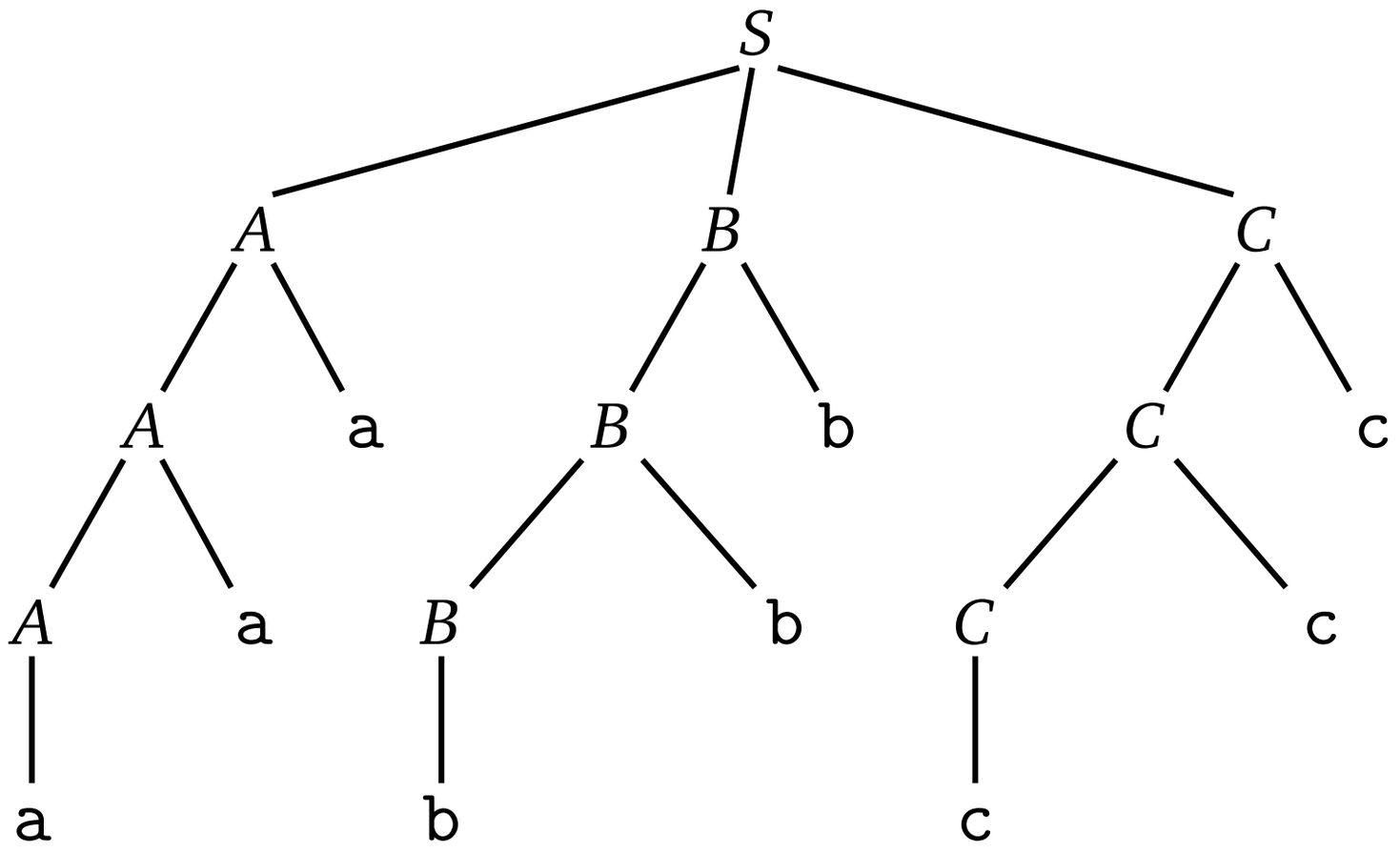
Inherited attributes: Attributes flow from left to right (from LHS to RHS and from symbols of RHS to symbols of RHS further to the right).

$$
\begin{array}{ll}
S \to ABC & \triangleright\ B.inhCount := A.count;\ C.inhCount := A.count \\
A \to \mathtt{a} & \triangleright\ A.count := 1 \\
A_1 \to A_2\ \mathtt{a} & \triangleright\ A_1.count := A_2.count + 1 \\
B \to \mathtt{b} & \triangleright\ \text{Condition: } B.inhCount = 1 \\
B_1 \to B_2\ \mathtt{b} & \triangleright\ B_2.inhCount := B_1.inhCount - 1 \\
C \to \mathtt{c} & \triangleright\ \text{Condition: } C.inhCount := 1 \\
C_1 \to C_2\ \mathtt{c} & \triangleright\ C_2.inhCount := C_1.inhCount - 1
\end{array}
$$

Inherited Attributes: Parse Tree Decoration (1)

Input: `aaabbbccc`

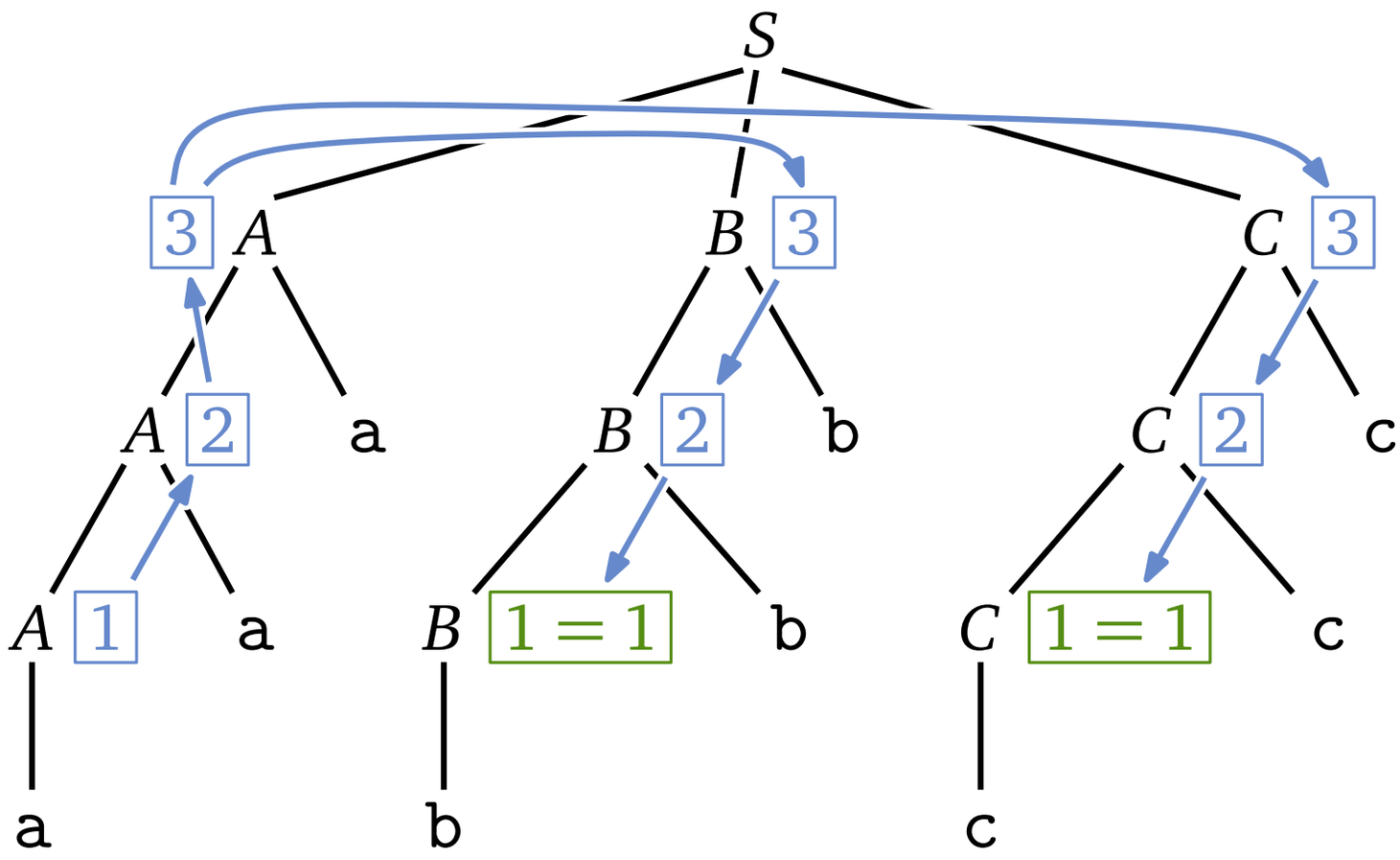
Parse tree



Inherited Attributes: Parse Tree Decoration (1)

Input: `aaabbbccc`

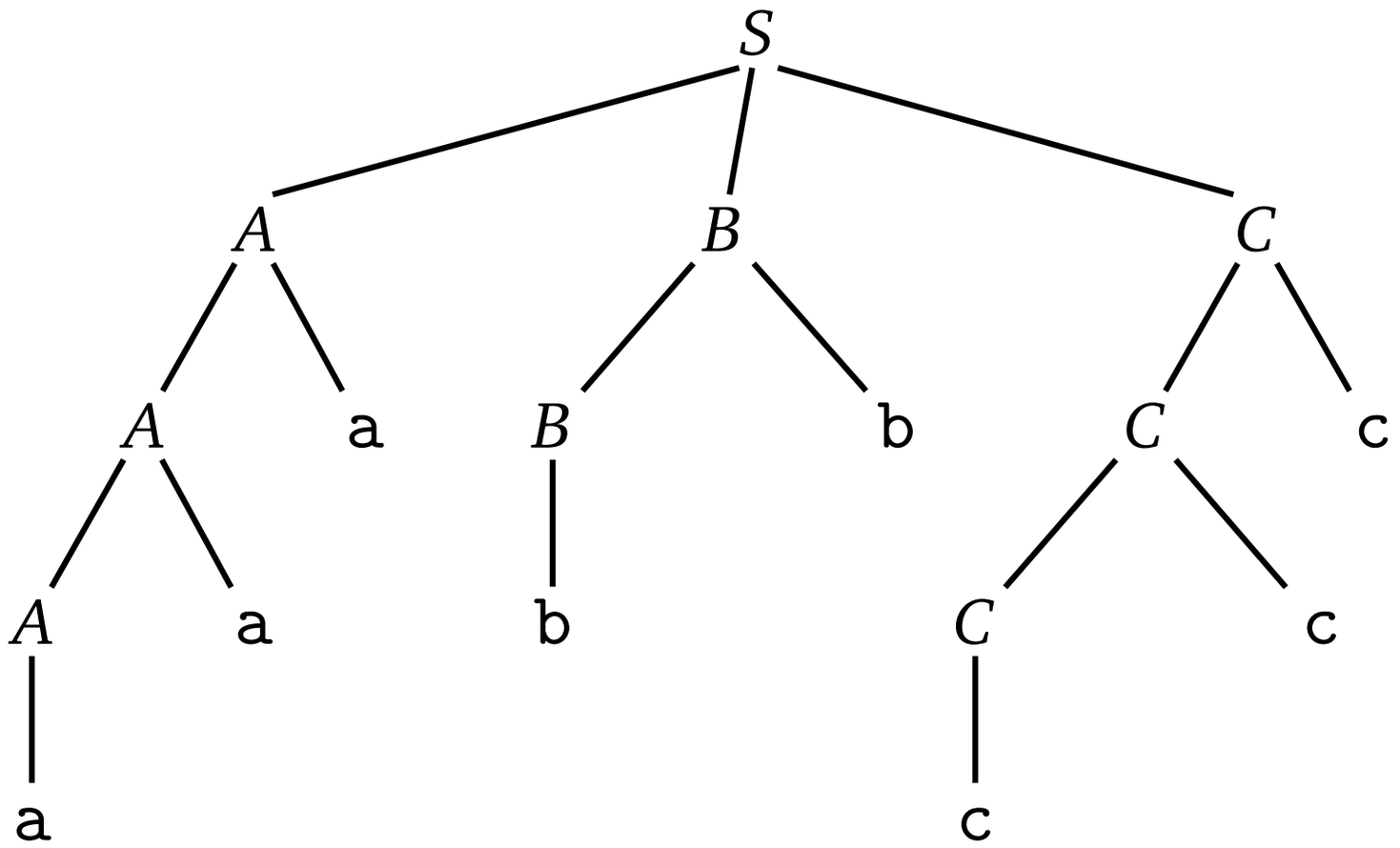
Parse tree



Inherited Attributes: Parse Tree Decoration (2)

Input: `aaabbccc`

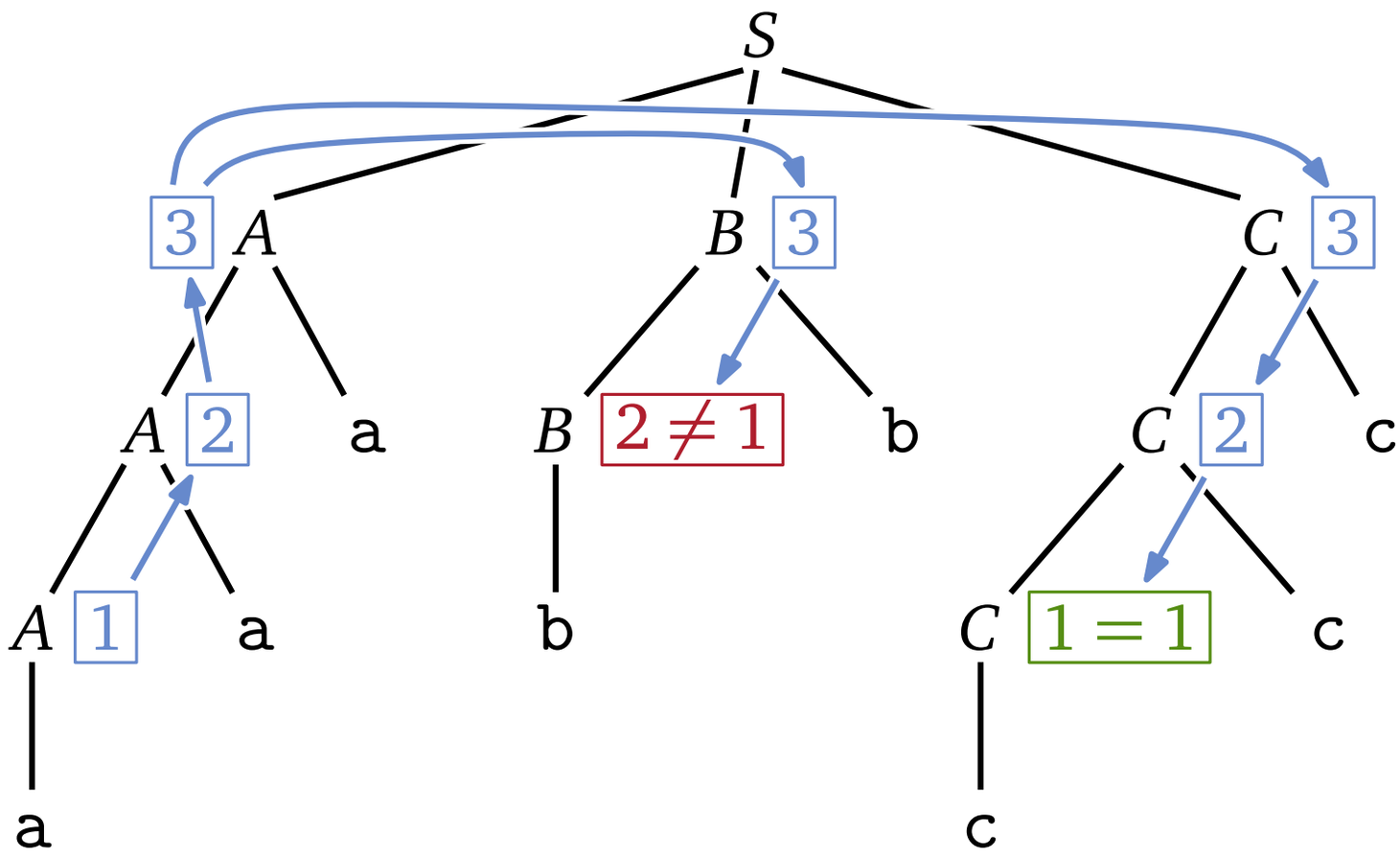
Parse tree



Inherited Attributes: Parse Tree Decoration (2)

Input: `aaabbccc`

Parse tree



Attribute Flow

Annotation or decoration of parse tree

- Process of evaluating attributes

Synthesized attributes

- Attributes of LHS of a production are computed from attributes of the symbols in its RHS.
- Attributes flow bottom-up in the parse tree.

Inherited attributes

- Attributes for symbols in RHS are computed from attributes of LHS and symbols in RHS preceding them.
- Attributes flow top-down in the parse tree.

S-Attributed and L-Attributed Grammars

S-attributed grammar

- All attributes are synthesized.
- Attributes flow bottom-up.

L-attributed grammar

- For each production $X \rightarrow Y_1 Y_2 \ldots Y_k$,
  - $X.syn$ depends on $X.inh$ and $Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, \ldots, Y_k.inh, Y_k.syn$.
  - For all $1 \leq i \leq k$, $Y_i.inh$ depends on $X.inh$ and $Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, \ldots, Y_{i-1}.inh, Y_{i-1}.syn$.

S-attributed grammars are a special case of L-attributed grammars.

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

A simple grammar for arithmetic expressions using only addition and subtraction

$$10 - 3 + 5 = (10 - 3) + 5 = 12$$

$$10 - 3 + 5 \neq 10 - (3 + 5) = 2$$

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

A simple grammar for arithmetic expressions using only addition and subtraction

$E \rightarrow T$

$E \rightarrow EAT$

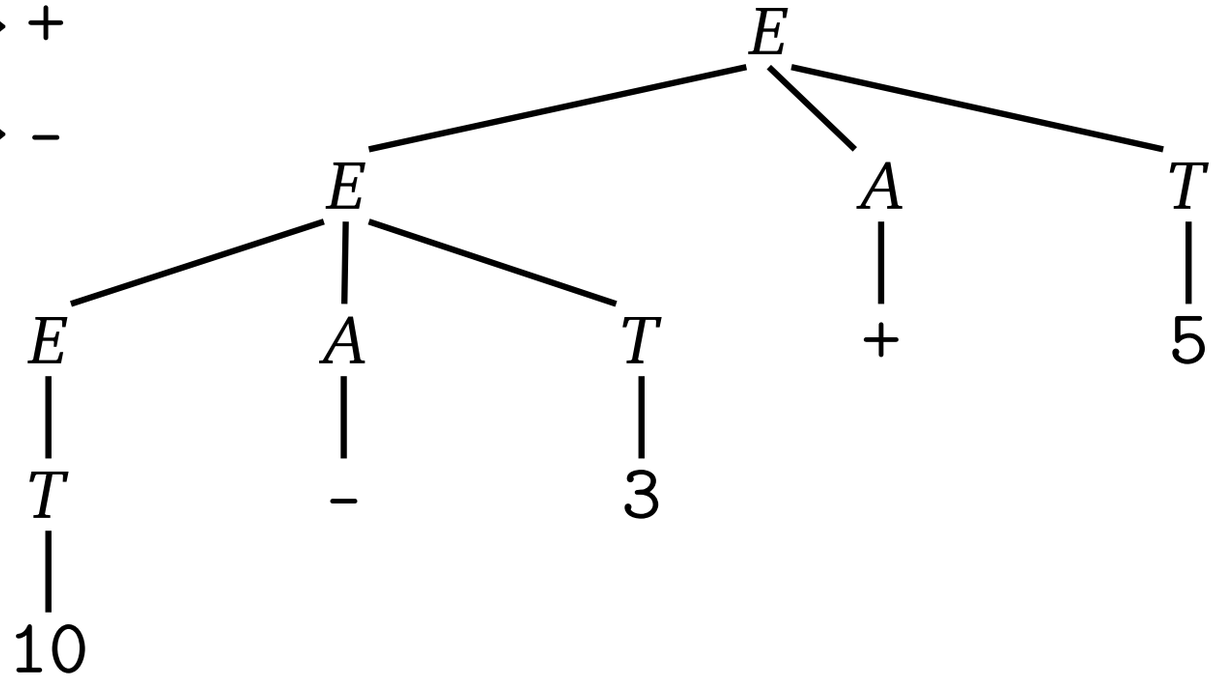
$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$$10 - 3 + 5 = (10 - 3) + 5 = 12$$

$$10 - 3 + 5 \neq 10 - (3 + 5) = 2$$



LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

A simple grammar for arithmetic expressions using only addition and subtraction

$E \rightarrow T$

$E \rightarrow EAT$

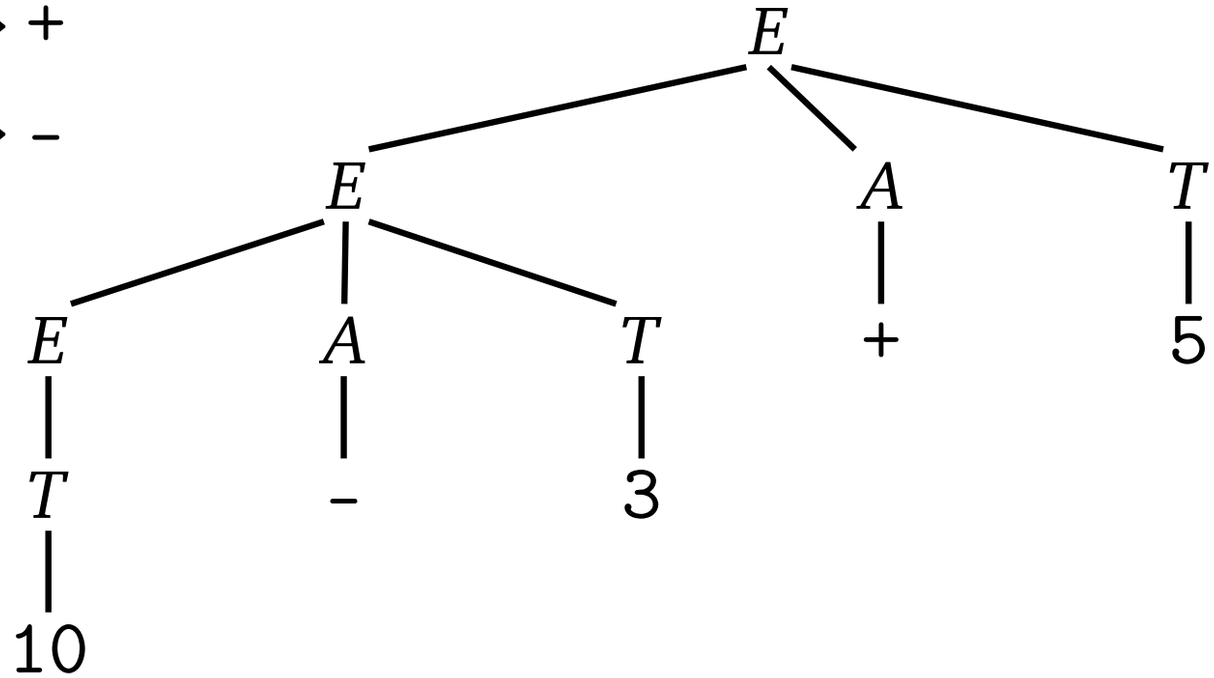
$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$$10 - 3 + 5 = (10 - 3) + 5 = 12$$

$$10 - 3 + 5 \neq 10 - (3 + 5) = 2$$



This grammar captures left-associativity.

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

A simple grammar for arithmetic expressions using only addition and subtraction

$E \rightarrow T$

$E \rightarrow EAT$

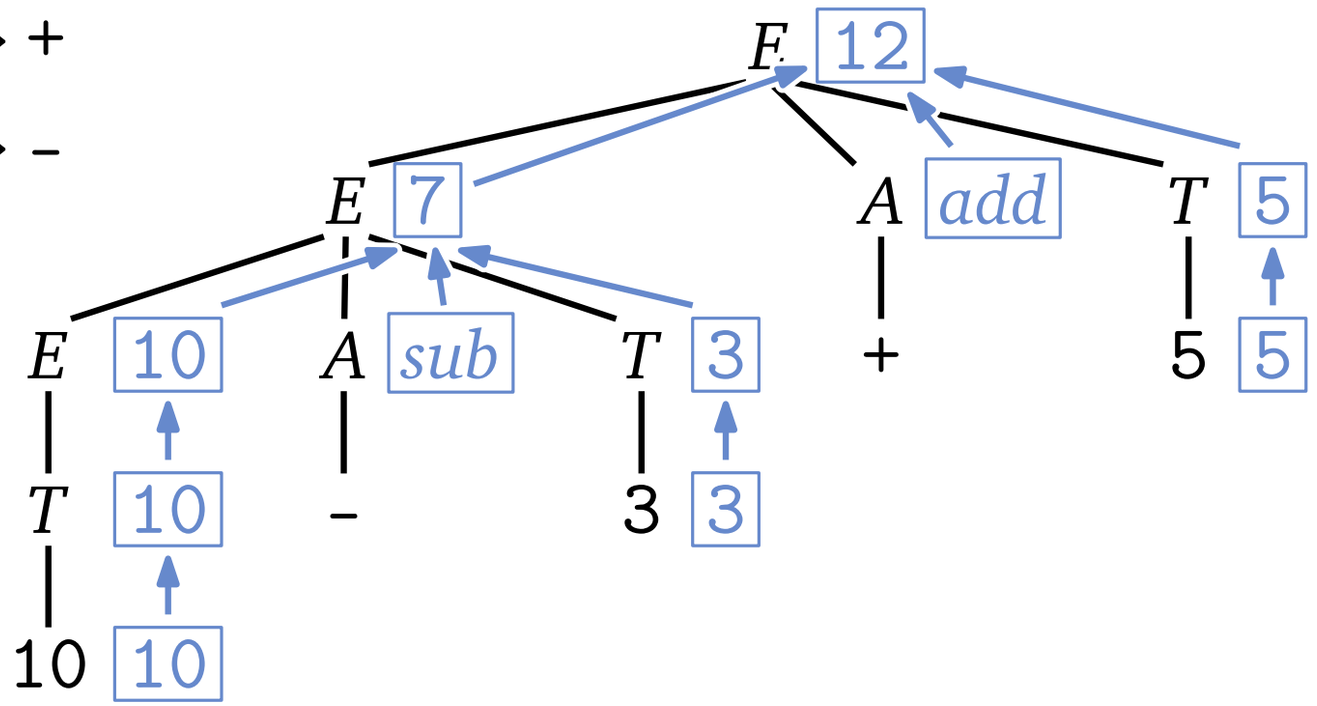
$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$$10 - 3 + 5 = (10 - 3) + 5 = 12$$

$$10 - 3 + 5 \neq 10 - (3 + 5) = 2$$



This grammar captures left-associativity.

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

A simple grammar for arithmetic expressions using only addition and subtraction

$E \to T$

$E \to EAT$

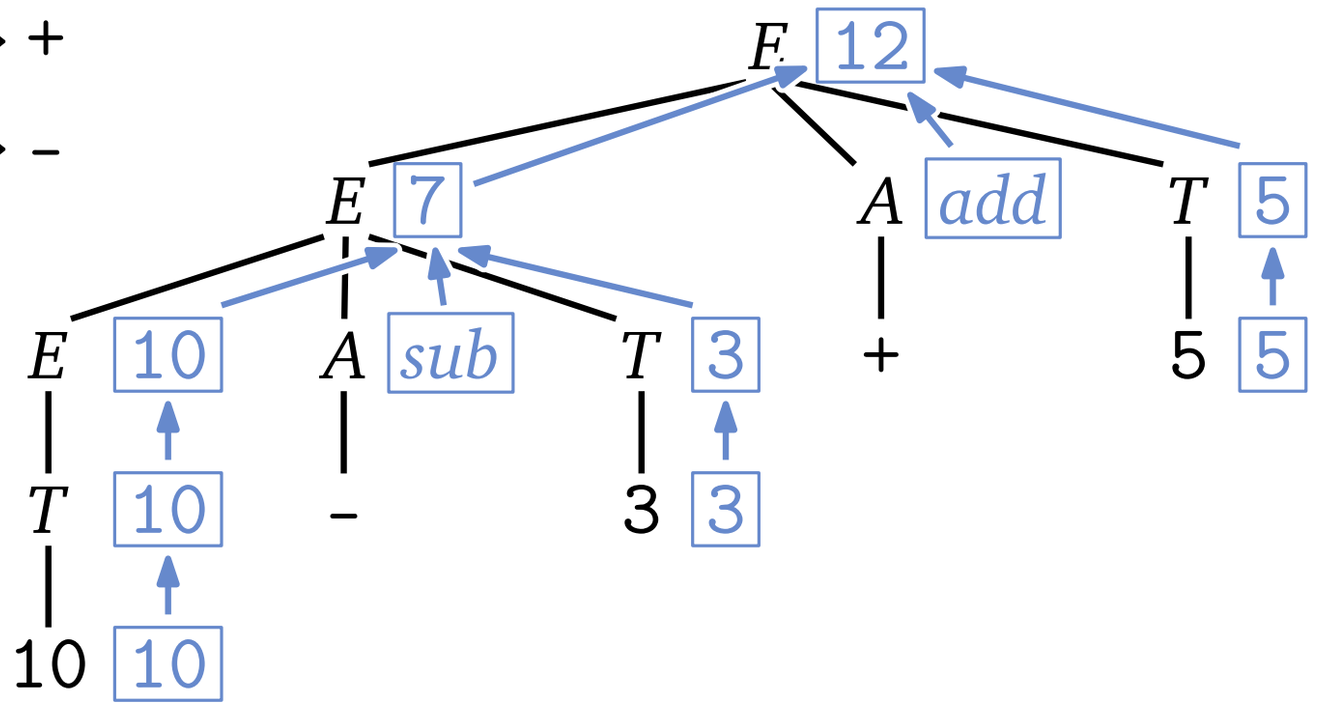
$T \to n$

$A \to +$

$A \to -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



Rule R	PREDICT(R)
$E \to T$	$\{n\}$
$E \to EAT$	$\{n\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$

This grammar captures left-associativity.

It is not LL(1).

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

An LL(1) grammar for the same language

$$
\begin{aligned}
E &\to TE'\$ \\
E' &\to \varepsilon \\
E' &\to ATE' \\
T &\to n \\
A &\to + \\
A &\to -
\end{aligned}
$$

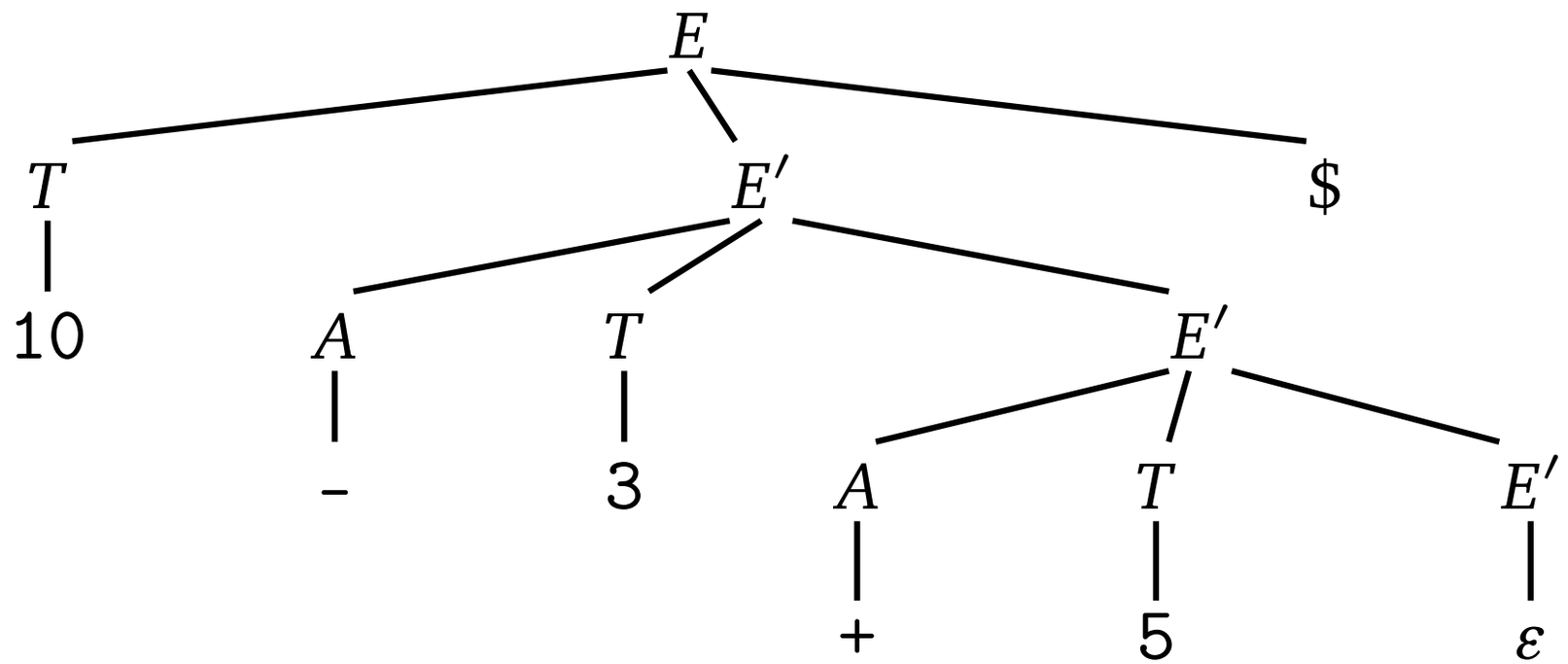
Rule R	PREDICT(R)
$E \to TE'\$$	$\{n\}$
$E' \to \varepsilon$	$\{\$\}$
$E' \to ATE'$	$\{+, -\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$

LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

An LL(1) grammar for the same language

$$E \to TE'\$$$
$$E' \to \varepsilon$$
$$E' \to ATE'$$
$$T \to n$$
$$A \to +$$
$$A \to -$$

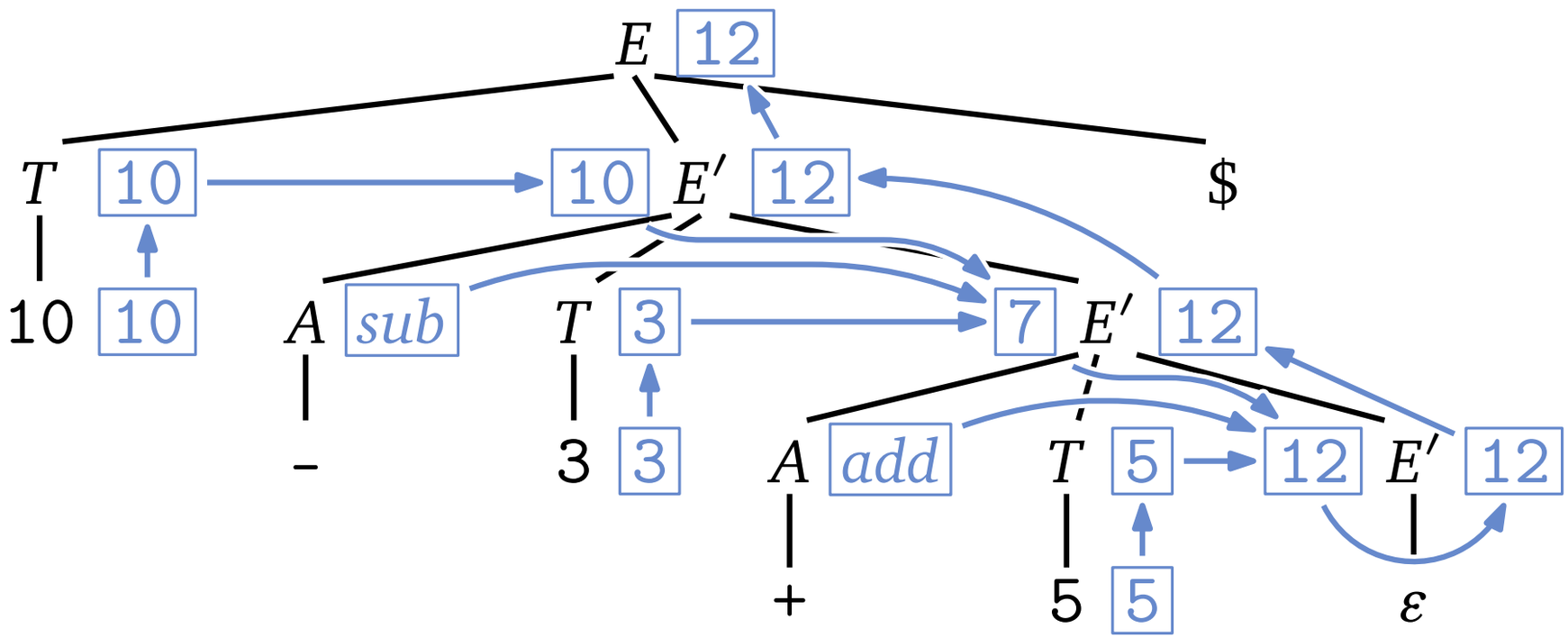
Rule R	PREDICT(R)
$E \to TE'\$$	$\{n\}$
$E' \to \varepsilon$	$\{\$\}$
$E' \to ATE'$	$\{+, -\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$



LL(1) Parsing, Left-Associativity, and L-Attributed Grammars

An LL(1) grammar for the same language

$$
\begin{aligned}
E &\to T\,E'\,\$ && \triangleright\ E'.op := T.val;\ E.val := E'.val \\
E' &\to \varepsilon && \triangleright\ E'.val := E'.op \\
E'_1 &\to A\,T\,E'_2 && \triangleright\ E'_2.op := A.fun(E'_1.op, T.val);\ E'_1.val := E'_2.val \\
T &\to n && \triangleright\ T.val := n.val \\
A &\to \texttt{+} && \triangleright\ A.fun := \texttt{add} \\
A &\to \texttt{-} && \triangleright\ A.fun := \texttt{sub}
\end{aligned}
$$



Action Routines

Action routines are instructions for ad-hoc translation interleaved with parsing.

Parser generators allow programmers to specify action routines in the grammar.

Action routines can appear anywhere in a rule (as long as the grammar is LL(1)).

Example

$$E'_1 \rightarrow A\ T\ \{\ E'_2.op = A.fun(E'_1.op, T.val)\ \}\ E'_2\ \{\ E'_1.val = E'_2.val\ \}$$

Action routines are supported, for example, in `yacc` and `bison`.