An ambiguous expression grammar:

$$\langle E\rangle ::= \langle E\rangle - \langle E\rangle \mid \langle E\rangle * \langle E\rangle \mid (\langle E\rangle) \mid \mathbf{num} \qquad (G_1)$$

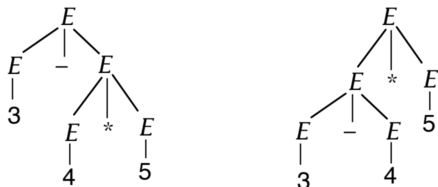
Two derivations:

$E \rightarrow E - \underline{E}$
$\quad \rightarrow \underline{E} - E * E$
$\quad \rightarrow \ldots$

$E \rightarrow \underline{E} * E$
$\quad \rightarrow \underline{E} - E * E$
$\quad \rightarrow \ldots$

Two expressions, same string.

Their parse trees:



...and their abstract syntax trees

This ambiguity could be removed by either

- adding a disambiguating rule or
- revising the grammar

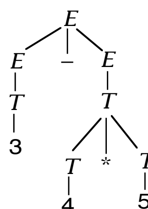
We revise the grammar:

$$\langle E\rangle ::= \langle E\rangle - \langle E\rangle \mid \langle T\rangle \qquad (G_2)$$
$$\langle T\rangle ::= \langle T\rangle * \langle T\rangle \mid (\langle E\rangle) \mid \mathbf{num}$$

Now the only tree that corresponds to

3 – 4 * 5

is



The revised grammar G_2 gives (*) a higher precedence than (–).

But G_2 is still ambiguous: Consider the expression

3 – 4 – 5

which can be derived two ways:

$E \rightarrow \underline{E} - E$
$\quad \rightarrow 3 - \underline{E}$
$\quad \rightarrow 3 - \underline{E} - E$
$\quad \rightarrow 3 - 4 - \underline{E}$
$\quad \rightarrow 3 - 4 - 5$

$E \rightarrow E - \underline{E}$
$\quad \rightarrow \underline{E} - 5$
$\quad \rightarrow T - \underline{E} - 5$
$\quad \rightarrow \underline{E} - 4 - 5$
$\quad \rightarrow 3 - 4 - 5$

3 – (4 – 5)
= 3 – (–1)
= 4

(3 – 4) – 5
= –1 – 5
= – 6

In other words, (–) can be either **right**-associative or **left**-associative —and so can (*).

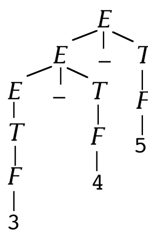
Again we can revise the grammar, to either

$$\langle \mathbf{E}\rangle ::= \langle \mathbf{E}\rangle - \langle T\rangle \mid \langle T\rangle \qquad (G_3)$$
$$\langle \mathbf{T}\rangle ::= \langle \mathbf{T}\rangle * \langle F\rangle \mid \langle F\rangle \qquad \textbf{left}\text{-recursive}$$
$$\langle F\rangle ::= (\langle E\rangle) \mid \mathbf{num}$$

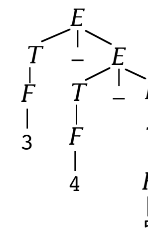
or

$$\langle \mathbf{E}\rangle ::= \langle T\rangle - \langle \mathbf{E}\rangle \mid \langle T\rangle \qquad (G_4)$$
$$\langle \mathbf{T}\rangle ::= \langle F\rangle * \langle \mathbf{T}\rangle \mid \langle F\rangle \qquad \textbf{right}\text{-recursive}$$
$$\langle F\rangle ::= (\langle E\rangle) \mid \mathbf{num}$$

Using G_3:



Using G_4:



So...

left-recursion yields *left*-associativity
right-recursion yields *right*-associativity.

Alternative grammar notations

EBNF (Extended BNF) —one example:

more meta-notation ⇒ shorter productions

- nonterminals begin with Uppercase letters (hence there's no need for ⟨ ⟩)
- grouping uses ( )
- alternatives are separated by |
- repetitions (zero or more) are enclosed in { } (replacing many recursions)
- options are enclosed in [ ]
- terminals that are also grammar symbols (for example, '[') are enclosed in single quotes ' '

Example: integer expressions again

<u>*Expression*</u>	::=	*Term* { (+ | -) *Term* }
Term	::=	*Factor* { (* | **/**) *Factor* }
Factor	::=	'(' *Expression* ')' | *Name* | *Numeral*
Name	::=	*Letter* { *Letter* | *Digit* }
Numeral	::=	*Digit* { *Digit* }
Letter	::=	a | b | … | z | 'A' | … | 'Z'
Digit	::=	0 | 1 | 2 | … | 9

How is operators' associativity specified in an EBNF grammar?

Example:

E ::= *T* { + *T* }

Does this make (+) left-associative or right-associative?

Answer: *Yes* (i.e., it's ambiguous).

The {…} notation obscures operators' associativity, so a grammar using it may need an "extra-syntactic" rule to eliminate the ambiguity.

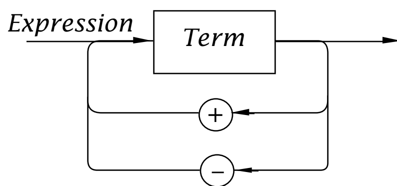
The conventional associativity rule for EBNF:

E ::= *T* { + *T* }	is understood to mean that (+) is *left*-associative
E ::= { *T* + } *T*	is understood to mean that (+) is *right*-associative

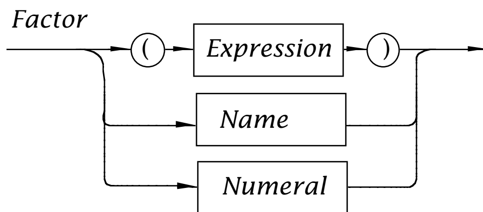
The ambiguity can be avoided by using recursion:

E ::= [*E* +] *T*	(+) is *left*-associative
E ::= *T* [+ *E*]	(+) is *right*-associative

Syntax charts ("railroad diagrams")

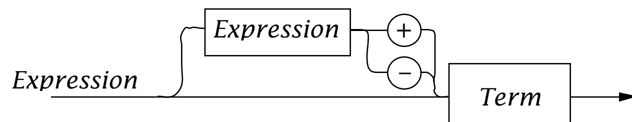


Each production ⇒ a path through the chart.



…etc.

How could syntax charts show associativity?



Context-*sensitive* grammars

In a C**F**G, selection among a nonterminal's productions is unconstrained, but in a C**S**G, the selection may depend on the symbols on either side of the nonterminal.

Example of a context-*sensitive* grammar:

S ::= **a b c** | **a** *X* **b c**

X **b** ::= **b** *X*

X **c** ::= *Y* **b c c**

a *Y* ::= **a a** | **a a** *X*

b *Y* ::= *Y* **b**

Note the terminals appearing in some productions' left-hand sides.

A derivation:

S → **a** <u>*X* **b**</u> **c**
→ **a b** <u>*X* **c**</u>
→ **a** <u>**b** *Y*</u> **b c c**
→ <u>**a** *Y*</u> **b b c c**
→ **a a b b c c**

This grammar generates a language of sequences containing equal numbers of **a**'s, **b**'s, and **c**'s, in that order. It has been shown that no CFG can produce this language.

Although CSGs are more powerful than CFGs, this power has not been found useful for defining the syntax of programming languages.