Operator precedence. Operator precedence specifies the manner in which operands are grouped with operators. For example, $1 + 2 \times 3$ is treated as $1 + (2 \times 3)$, whereas $1 \times 2 + 3$ is treated as $(1 \times 2) + 3$ because the multiplication operator has a higher precedence than the addition operator. You can use parentheses to override the default operator precedence rules.
Operator associativity. When an expression has two operators with the same precedence, the operators and operands are grouped according to their associativity. For example, $72 \div 2 \div 3$ is treated as $(72 \div 2) \div 3$ since the division operator is left-to-right associative. You can use parentheses to override the default operator associativity rules.

Operators may be associative (meaning the operations can be grouped arbitrarily), left-associative (meaning the operations are grouped from the left), right-associative (meaning the operations are grouped from the right) or non-associative (meaning operations cannot be chained, often because the output type is incompatible with the input types).
Associativity and Precedence examples

In order to reflect normal usage, addition, subtraction, multiplication, and division operators are usually left-associative.

\[
(1-2)-3 = -4 \\
1-(2-3) = 2 \\
2*(3+4) = 14 \\
(2*3)+4 = 10 \\
\]

\[
(2^3)^4 = 4096 \\
2^{(3^4)} = 2417851639229258349412352 \\
\]

for an exponentiation operator there is no general agreement.
Consider we would like to create a grammar for basic arithmetic operations like 1+2+3 or 7/8+9-2

```
BINARY_OP: "+" | "-" | "*" | "/"
expr : expr BINARY_OP expr
     | NUMBER
```
BINARY_OP: "+" | "-" | "*" | "/"
expr : expr BINARY_OP expr
     | NUMBER

One interpretation: \((1+2)+3\times3=6\)

Another interpretation: \((1+(2\times3))=1+5=6\)
One interpretation, correct: \((1-2)-3=-1-3=-4\)

Another interpretation, incorrect: \(1-(2-3)=1-(-1)=2\)
We control left/right associativity by selecting which side of the operator recursion is performed

```
BINARY_OP: "+" | "-" | "*" | "/"
expr : expr BINARY_OP NUMBER
     | NUMBER
```

Now we have only one possibility (left-asoc subtraction)
In the formal language theory of computer science, left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right). For instance, $1 + 2 + 3$ can be recognized as a sum because it can be broken into $1 + 2$, also a sum, and $+ 3$, a suitable suffix.

$$1+2+3+4$$

is an expression, $1+2$, followed by suitable suffix, $\text{expr} + 3+4$

Which is $\text{expr} + 4$

Which is $\text{expr}$
We control left/right associativity by selecting which side of the operator recursion is performed.

If we turn it another way around, operators would be right-associative.

BINARY_OP: "+" | "-" | "*" | "/"

expr : NUMBER BINARY_OP expr
| NUMBER

If we switch recursion to right, operator is right-asoc
Grammar is unambiguous but incorrect

\[ 1+2*3 = (1+2)*3 \]
First, we need to select how many priority levels we have. For now, there are two: */ before +-. For each level, we introduce a separate expression type. As */ needs to be computed before +-, we conclude that multiplicative expression should be “under” additive.

```
ADD_OP: "+" | "-"
MUL_OP: "*" | "/"

add_expr: add_expr ADD_OP mul_expr
          | mul_expr
          | mul_expr

mul_expr: mul_expr MUL_OP NUMBER
          | NUMBER

expr: add_expr
```
$1 + 2 \times 3 = 1 + (2 \times 3)$
the exponentiation operator \(^\) will be right-associative and has the highest priority.

\[
\begin{align*}
\text{ADD\_OP: } & \text{ "+" } \mid \text{ "-" } \\
\text{MUL\_OP: } & \text{ "*" } \mid \text{ "/" } \\
\text{POWER\_OP: } & \text{ "^" } \\
\text{add\_expr: } & \text{ add\_expr ADD\_OP mul\_expr} \mid \text{ mul\_expr} \\
\text{mul\_expr: } & \text{ mul\_expr MUL\_OP pow\_expr} \mid \text{ pow\_expr} \\
\text{pow\_expr: } & \text{ NUMBER POWER\_OP pow\_expr} \mid \text{ NUMBER} \\
\text{expr: } & \text{ add\_expr}
\end{align*}
\]
\[1 + 2^3 \times 4 \times 5 = 1 + 2 \times (3^4) \times 5 = 1 + (2^4) \times 5 = 1 + ((2^4) \times 5)\]
unary minus for expressions like -2+3 or 7*-5. we introduce a new non-terminal: primary. Primary can be either a number or a unary operator followed by a number.

ADD_OP: "+" | "-
MUL_OP: "+" | "/
POWER_OP: "^"

add_expr: add_expr ADD_OP mul_expr
          | mul_expr

mul_expr: mul_expr MUL_OP pow_expr
          | pow_expr

pow_expr: primary POWER_OP pow_expr
         | primary

primary: "-" NUMBER
        | NUMBER

expr: add_expr
-2+3 = (-2)+3
By definition, an expression in parenthesis has the highest priority.
\( 2^\left(3 - 1\right) \)