An expression consists of

- An atomic object, e.g. number or variable.
- An operator (or function) applied to a collection of operands (or arguments) each of which are also expressions.

Note: conventionally, the term operator refers to a built-in function with special syntax. For example “a + b”, might really stand for a function call like `sum(a, b)` or `a.sum(b)`. 
Common syntactic forms for operators:

- **Function call notation**, e.g. `somefunc(A, B, C)`
- **Infix notation** for binary operators, e.g. `A + B`
  - Short for “+”(A, B) in Ada and `A.operator+(B)` in C++.
- **Prefix notation** for unary operators, e.g. `-A`
- **Postfix notation** for unary operators, e.g. `i++`
- **Cambridge Polish notation** (Lisp)
  - Prefix notation, function inside parentheses: `(* (+ 1 3) 2)`
- "**Multi-word**" infix, e.g. `a > b ? a : b` in C
The use of infix notation sometimes leads to ambiguity as to what operands an operation should be applied to.

**Fortran example:** \( a + b \times c^{d^{e/f}} \)

- \( (((a+b) \times c)^{d^{e/f}}) \)? or...
- \( a + (((b \times c)^{d^{e/f}}) \times e/f) \)? or...
- \( a + ((b \times (c^{d^{e/f}})) \times e/f) \)?

A programming language must be able to disambiguate such expressions. Two options:

- Add parentheses.
- Define operator precedence and associativity in the language that specify the order of evaluation in the absence of parentheses.
**PRECEDENCE AND ASSOCIATIVITY**

- **Operator precedence**: higher operator precedence means that a collection of operators group more tightly in an expression than operators of lower precedence.

- **Operator associativity**: determines grouping of operators of the same precedence.
  - Left associative: operators are grouped left-to-right (most common).
  - Right associative: operators are grouped right-to-left (Fortran power operator **, C assignment operator = and unary minus).
  - Non-associative: requires parentheses when composed (Ada power operator **).

In Fortran: ** > { *, / } > { +, - } and +, -, *, / are left associative and ** is right associative.

\[ a + b * c ** d ** e / f = ? \]
PRECEDENCE AND ASSOCIATIVITY

• In a language, the number of operators can be quite large.
  • E.g. C/C++ has around 18 different precedence levels.
  • On the other hand, Pascal has about 4.
  • The programmer of a language must be careful about the precedence and associativity.

• In C/C++,
  • if (A < B && C < D) {...} = if ((A < B) && (C < D)) {...}

• In Pascal,
  • if A<B and C<D then = if A<(B and C)<D then
  • and is of higher precedence than equality comparisons.
EVALUATION ORDER

• Precedence and associativity state the rules for grouping operators in expressions, but do not determine the operand evaluation order!
  
  • Expression $a - f(b) - b * c$
  
  is evaluated as $(a - f(b)) - (b * c)$
  
  but either $(a - f(b))$ or $(b * c)$ could be evaluated first…

• The evaluation order of arguments in subroutine calls may differ, e.g. arguments evaluated from left to right or right to left.

• Knowing the operand evaluation order is important.
  
  • Side effects: suppose $f(b)$ above modifies the value of $b$ ($f(b)$ has a “side effect”), then the value will depend on the operand evaluation order.
  
  • Code improvement: compilers rearrange expressions to maximize efficiency, e.g. a compiler can improve memory load efficiency by moving loads up in the instruction stream. We may want to evaluate $f(b)$ first so that we don’t need to save $b*c$ during its execution.
Some languages allow the compiler to rearrange mathematical expressions in accordance with commutative, associative, or distributive laws in order to gain some improvement in efficiency. For example,

\[
\begin{align*}
a &= b + c \\
d &= c + e + b
\end{align*}
\]

may be rewritten as

\[
\begin{align*}
a &= b + c \\
d &= b + c + e
\end{align*}
\]

which allows the common subexpression to be reused in the intermediate code.
EXPRESSION OPERAND REORDERING ISSUES

Rearranging expressions may lead to arithmetic overflow or different floating point results.

• Assume $b$, $d$, and $c$ are very large positive integers, then if $b-c+d$ is rearranged into $(b+d) - c$ arithmetic overflow occurs.

• Floating point value of $b-c+d$ may differ from $b+d-c$.

• Most programming languages will not rearrange expressions when parentheses are used, e.g. write $(b-c)+d$ to avoid problems.

Design choices

• Java: expression evaluation is always left to right in the order operands are provided in the source text and overflow is always detected.

• Pascal: expression evaluation is unspecified and overflows are always detected.

• C and C++: expression evaluation is unspecified and overflow detection is implementation dependent.
Boolean expressions are not quite the same as the regular arithmetic expressions.

• To evaluate an arithmetic expression, all components must be evaluated.
To evaluate \((a + b) - (c + d)\): compute \(a + b\), compute \(c + d\), compute \((a + b) - (c + d)\).

  \[
  \begin{align*}
  T1 &= a+b \\
  T2 &= c+d \\
  T3 &= T1-T2 \\
  \end{align*}
  \]

• To evaluate a boolean expression, we may or may not need to evaluate all components.
Consider \((a < b) \&\& (c < d)\).

Option 1 (do it like regular arithmetic expression):

  \[
  \begin{align*}
  T1 &= a<b \\
  T2 &= c<d \\
  T3 &= T1 \&\& T2 \\
  \end{align*}
  \]
SHORT-CIRCUIT EVALUATION

To evaluate a boolean expression, we may or may not need to evaluate all components. Consider $(a<b) \land (c<d)$.

Option 2 (do it without evaluating the whole thing):

\[
\begin{align*}
T1 &= a<b \\
\text{if } (T1 == \text{false}) & \text{ goto 100} \\
T2 &= c<d \\
T3 &= T1 \land T2 \\
goto 200
\end{align*}
\]

100: $T3 = \text{false}$

200: ...

Compute the boolean value for $a<b$. If false, we’re done (the whole expression is false). Otherwise, compute the boolean value for $c<d$, etc. Is this how C/C++ works?
Computing the Boolean value of an expression without evaluating the whole expression (option 2) is called short-circuit evaluation.

With short-circuit evaluation of Boolean expressions, the result of an operator can be determined from the evaluation of just one operand.

C, C++, and Java use short-circuit conditional and/or operators.

• If \( a \text{ in } a \&\& b \) evaluates to false, \( b \) is not evaluated (\( a \ &\& \ b \) is false).

• If \( a \text{ in } a \mid \mid b \) evaluates to true, \( b \) is not evaluated (\( a \ |\ | \ b \) is true).
SHORT-CIRCUIT EVALUATION

Consider these examples. Why might short-circuit evaluation be so important here?

- $\text{for } (i=0; (i<N) \land (a[i] \neq \text{val}); i++) \ldots$;
- $\text{while } ((p\neq \text{NULL}) \land (p->\text{value} \neq \text{val})) \ p=p->\text{next}$;
ASSIGNMENTS AND EXPRESSIONS

The use of expressions and assignments is the fundamental difference between imperative and functional languages.

• Imperative: "computing by means of side effects".
  • Computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing of values of variables.

• Pure functional languages: computation consists entirely of expression evaluation.
  • A function always returns the same value given the same arguments because of the absence of side-effects (no memory state is changed implicitly in such a function).
L-VALUES VS. R-VALUES

Consider the assignment of the form: \( a = b + c \)

The left-hand side of the assignment is an L-value which is an expression that should denote a location.
  • e.g. array element \( a[2] \) or a variable \( \text{foo} \) or a dereferenced pointer \( \ast p \).

The right-hand side of the assignment is an R-value which denotes the value.
  • e.g. an expression composed of variables and/or literals.
There are two general ways to implement an assignment.

- Languages that adopt the value model of variables copy the value of $b+c$ into the location of $a$ (e.g. Ada, Pascal, C). That is, $a$ is synonymous with some address in memory.

- Languages that adopt the reference model of variables copy references, resulting in shared data values via multiple references.

$$
\begin{align*}
B &= 2 \\
C &= B \\
A &= B+C
\end{align*}
$$

$$
\begin{align*}
A &= 4 \\
B &= 2 \\
C &= 2
\end{align*}
$$

A  \rightarrow  4

B  \rightarrow  2

C  \rightarrow  2