Lateralus Language Specification v1.0

Formal grammar, type system, and standard library for Lateralus v1.0

Lateralus Language

bad-antics · December 2023 · Lateralus Language Research

ABSTRACT This document defines the initial formal grammar, primitive and compound types, function definitions, pipeline operator (|>) semantics, pattern matching syntax, and the minimal standard library for Lateralus v1.0.

1. Document Scope and Conventions

This specification is the normative reference for Lateralus v1.0. It defines what programs are syntactically valid, what types they may have, and what they mean when executed. Any conforming implementation must accept every program the grammar permits and must assign those programs the semantics described in sections 7 through 14.

Grammar productions are written in extended Backus-Naur Form (EBNF). Terminals appear in fixed-width type. The symbol ::= separates a non-terminal from its definition; | separates alternatives; * and + are Kleene star and plus; ? is optional.

Type judgments are written in natural-deduction style: hypotheses appear above a horizontal rule and the conclusion below. The typing environment is denoted Gamma; the turnstile |- reads 'entails'. Metavariables: e for expressions, t for types, x for identifiers.

Normative requirements use the keyword **MUST**; guidance uses **SHOULD**; permissions use **MAY**. Sections marked **(informative)** are not normative but provide rationale. All section cross-references are to sections within this document unless otherwise noted.

v1.0 targets a single-pass tree-walking interpreter. No multi-file compilation, no separate linking step, and no unsafe blocks exist in v1.0. Subsequent versions extend this baseline; this document is the stable reference for v1.0 only.

2. Lexical Grammar

The lexer consumes a UTF-8 byte stream and emits a flat token sequence. Whitespace (space, tab, carriage return, newline) is insignificant except as a token separator and inside string literals. Comments begin with # and extend to the end of the line; they produce no tokens.

2.1 Identifier and Keyword Tokens

An identifier matches the pattern [a-zA-Z_][a-zA-Z0-9_]*. The following identifiers are reserved as keywords and may not be used as user-defined names:

```
fn       let      if       else     match    case     return
enum     import   pub      use      true     false    and
or       not      in       for      while    break    continue
```

2.2 Literals

Integer literals are one or more ASCII digits, optionally prefixed with 0x for hexadecimal or 0b for binary. Float literals contain exactly one decimal point and match [0-9]+\.[0-9]+. String literals are delimited by double quotes and support the escape sequences \n \t \\ \" \0. Boolean literals are true and false.

2.3 Operator Tokens

```
|>   |?>   +    -    *    /    %    ==   !=   <    <=   >   >=
=    ->   =>   ::   ,    ;    :    .    (    )    [    ]   {   }
```

3. Expression Grammar (BNF)

The expression grammar is presented in order of decreasing precedence. Lower-numbered levels bind more tightly. Section 10 lists the complete precedence table.

```
expr        ::= pipe_expr
pipe_expr   ::= cmp_expr (('|>' | '|?>') cmp_expr)*
cmp_expr    ::= add_expr (('==' | '!=' | '<' | '<=' | '>' | '>=') add_expr)*
add_expr    ::= mul_expr (('+' | '-') mul_expr)*
mul_expr    ::= unary_expr (('*' | '/' | '%') unary_expr)*
unary_expr  ::= ('-' | 'not') unary_expr | call_expr
call_expr   ::= primary_expr ('(' arg_list? ')')*
primary_expr::= literal | ident | '(' expr ')' | block | match_expr
              | list_expr | if_expr
block       ::= '{' stmt* '}'
arg_list    ::= expr (',' expr)*
list_expr   ::= '[' arg_list? ']'
```

The grammar is LL(1) after disambiguation: if and match are keywords consumed by primary_expr; block ambiguity is resolved by requiring that a statement-level block must follow a keyword.

A call_expr is left-recursive in the grammar above but the parser flattens this into a loop: each (arg_list) suffix wraps the accumulated callee in a new CallNode. Chained calls f(a)(b) are valid.

The expression grammar does not include assignment; assignment is a statement-level construct (Section 4). An expression used as a statement is followed by an optional semicolon; omitting the semicolon on the last expression in a block makes that expression the block's value.

4. Statement Grammar

Statements are the top-level structural unit within a block. A program is a sequence of top-level declarations (Section 5); there is no implicit top-level block in v1.0.

```
stmt        ::= let_stmt | return_stmt | while_stmt | for_stmt
              | break_stmt | continue_stmt | expr_stmt
let_stmt    ::= 'let' ident (':' type)? '=' expr ';'
return_stmt ::= 'return' expr? ';'
while_stmt  ::= 'while' expr block
for_stmt    ::= 'for' ident 'in' expr block
break_stmt  ::= 'break' ';'
continue_stmt::= 'continue' ';'
expr_stmt   ::= expr ';'?
```

A let_stmt introduces a new binding in the current scope. The optional type annotation is checked against the inferred type of the initializer expression; a mismatch is a compile-time error. Shadowing a name from an outer scope is permitted.

A for_stmt binds the loop variable to successive elements of the iterable expression. In v1.0 the iterable MUST be a list; iterator traits are a v2.0 feature. The loop variable is immutable within the loop body.

A return_stmt with no expression returns unit. Inside a function whose declared return type is not unit, a bare return; is a type error. At the top level (outside a function), return is not permitted and the parser MUST reject it.

5. Declaration Grammar

Declarations appear at module scope. In v1.0 the module scope is the file. A declaration is either a function definition, an enum declaration, an import, or a top-level let binding.

```
decl      ::= fn_decl | enum_decl | import_decl | let_decl
fn_decl   ::= 'fn' ident '(' param_list? ')' ('->' type)? block
param_list::= param (',' param)*
param     ::= ident ':' type
let_decl  ::= 'let' ident (':' type)? '=' expr ';'
enum_decl ::= 'enum' ident '{' variant (',' variant)* '}'
variant   ::= ident ('(' type_list ')')?
```

A function declaration introduces a name in module scope before any code runs; forward references within the same file are therefore valid. This is the only form of forward reference in v1.0; enum variants and module-level let bindings are evaluated in order.

A function without a declared return type implicitly returns the type of its last expression. If the function contains a return statement, all return paths MUST agree on a common type. The inference algorithm described in Section 13 resolves the type of recursive functions by initializing the return type to a fresh unification variable before descending into the body.

Enum variants without a payload have type EnumName directly. Variants with a payload are constructor functions of type t1 -> ... -> EnumName. The variant name is always qualified by the enum name at use sites: Option::Some(x), not Some(x).

6. Type Grammar

The v1.0 type grammar is intentionally small. Compound types are built from primitives by application.

```
type      ::= 'int' | 'float' | 'str' | 'bool' | 'unit'
            | ident                        # user-defined enum
            | type '[' type_list ']'       # generic application
            | '(' type_list ')' '->' type  # function type
            | '(' type ')'                 # grouping
type_list ::= type (',' type)*
```

Generic applications in v1.0 are limited to the standard library types list[T], option[T], and result[T, E]. User-defined generic enums are introduced in v2.0.

Function types are right-associative: (int) -> (int) -> int is (int) -> ((int) -> int). In v1.0, all functions are uncurried at the surface level; currying is achieved through explicit lambda-returning functions. Partial application is not automatic.

The type unit has exactly one value, written () at the expression level. It is used as the return type of functions that produce side effects only, and as the element type of option[unit] when signaling presence versus absence.

Recursive types are not expressible in v1.0 type annotations; the interpreter handles recursive values internally via its heap. A future version will expose nominal recursive types through rec bindings.

7. Function Definition Semantics

A function definition creates a closure over the environment at its definition site. In v1.0 all closures are immutable: the captured environment is snapshot at closure creation and cannot be mutated through the closure.

```
fn add(x: int, y: int) -> int {
    x + y
}

fn make_adder(n: int) -> (int) -> int {
    fn inner(x: int) -> int { x + n }
    inner
}
```

Parameters are passed by value. For primitive types (int, float, bool, str, unit) this is a shallow copy. For list values, it is a shallow copy of the list spine; the elements themselves are shared (copy-on-write semantics are deferred to v2.0). For enum values, it is a copy of the tag and all payload fields recursively.

A function may call itself recursively by name; the name is in scope within the function body for this purpose. Mutual recursion between top-level functions is supported because all top-level names are hoisted before evaluation begins. Mutual recursion between nested functions requires explicit forward declaration via a let binding initialized to an error-throwing stub, which is replaced by the real function.

Higher-order functions are fully supported: functions may be passed as arguments, returned from other functions, stored in lists, and bound with let. The type of a function value is its function type; the runtime representation is a closure record containing the code pointer and the captured environment.

8. The |> Pipeline Operator

The pipeline operator is the defining feature of Lateralus. It threads a value through a sequence of transformations without nesting function calls. Formally:

```
Reduction rule (|>):

    e1 |> e2

reduces to:

    e2(e1)

where e2 MUST evaluate to a function of arity >= 1.
```

The operator is left-associative and has lower precedence than comparison operators but higher precedence than logical and/or. A chain a |> f |> g |> h is parsed as ((a |> f) |> g) |> h and reduces to h(g(f(a))).

```
# Pipeline example: process a list of integers
let result =
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    |> std::list::filter(fn(x: int) -> bool { x % 2 == 0 })
    |> std::list::map(fn(x: int) -> int { x * x })
    |> std::list::sum;
# result: 220
```

When the right-hand side of |> is a partial application f(a, b) (not a plain identifier), the reduction is f(a, b, e1) -- the left-hand value is appended as the final argument. This convention means xs |> std::list::map(f) reads naturally as 'map f over xs'.

Type rule for the pipeline operator: if e1 : A and e2 : (A) -> B, then e1 |> e2 : B. The type checker unifies the first parameter type of e2 with the type of e1; a failure is reported as a pipeline type mismatch with both sides displayed.

9. The |?> Error Pipe

The error pipe |?> is a variant of |> designed for functions that return result[T, E]. It short-circuits on the result::Err variant, propagating the error without executing the remainder of the pipeline.

```
Reduction rule (|?>):

    e1 |?> e2

  case e1 = result::Ok(v)  => e2(v)
  case e1 = result::Err(e) => result::Err(e)   (short-circuit)
```

The containing function MUST have a return type of result[T2, E] where E is the same error type as the Err variant of e1. This is enforced by the type checker. In v1.0, the error type must match exactly; covariant error types are a v2.0 feature.

```
fn parse_and_double(s: str) -> result[int, str] {
    s
    |?> std::string::parse_int
    |?> fn(n: int) -> result[int, str] { result::Ok(n * 2) }
}
```

When a pipeline chain mixes |> and |?>, they can appear in any order. A |> step after a |?> receives the unwrapped OK value because the error already short-circuited. The programmer is responsible for re-wrapping the final value in result::Ok if the chain is expected to return a result type.

10. Arithmetic and Comparison Operators

The arithmetic operators + - * / % are defined on int and float. Mixed-type arithmetic (int + float) is a type error in v1.0; explicit conversion via std::math::int_to_float is required.

```
# Operator precedence (highest to lowest)
# Level 1:  unary  -  not
# Level 2:  *  /  %
# Level 3:  +  -
# Level 4:  ==  !=  <  <=  >  >=
# Level 5:  and
# Level 6:  or
# Level 7:  |>  |?>
# All binary operators are left-associative at the same level.
```

Integer division / performs truncating division toward zero. The remainder operator % satisfies (a / b) * b + (a % b) == a for all nonzero b. Division by zero is a runtime error that raises DivisionByZero; there is no static check in v1.0.

Comparison operators return bool. They are defined on int, float, and str (lexicographic). Comparing values of different types is a type error. Equality (==, !=) is additionally defined on bool and on enum values; structural equality for lists requires std::list::equal.

The logical operators and and or are short-circuit: the right operand is not evaluated if the result is determined by the left operand. They take and return bool; implicit coercion from other types is not supported.

11. Let Bindings and Scoping

Lateralus uses lexical (static) scoping throughout. A let binding introduces a new name in the innermost enclosing scope; the name is visible from immediately after the binding to the end of that scope.

```
let x = 10;
{
    let x = 20;    # shadows outer x
    let y = x + 1; # y = 21
}
# x = 10 here; y is not in scope
```

Bindings are immutable in v1.0. There is no let mut or reassignment statement. Loops that accumulate a result do so by passing an accumulator as a function argument or by building up a list and reducing it at the end.

The value semantics of v1.0 make immutability unambiguous: assigning a list to a new name copies the list spine. Two bindings can share elements (since elements are heap-allocated), but mutating one through a future mutable operation (v2.0) would not affect the other's spine.

Bindings introduced in the initializer of another binding are not in scope for that initializer. In other words, let x = x + 1 refers to the outer x, not a recursive self-reference. The exception is function declarations using fn, which are automatically recursive (Section 7).

12. Recursive Functions

Any function declared with fn at either module scope or local scope may call itself by name within its body. The name is bound before the body is entered, so the self-reference is always valid.

```
fn factorial(n: int) -> int {
    if n <= 1 { 1 } else { n * factorial(n - 1) }
}

fn fib(n: int) -> int {
    if n <= 1 { n } else { fib(n - 1) + fib(n - 2) }
}
```

Tail recursion is not optimized in v1.0. A program that recurses deeply enough will exhaust the call stack and produce a StackOverflow runtime error. The recommended workaround is to use explicit loops (while or for) for tail-recursive patterns. Tail-call optimization is planned for v1.1.

Mutually recursive top-level functions are supported without any special syntax because top-level names are hoisted. Two functions even and odd may call each other freely.

```
fn even(n: int) -> bool {
    if n == 0 { true } else { odd(n - 1) }
}
fn odd(n: int) -> bool {
    if n == 0 { false } else { even(n - 1) }
}
```

13. Algebraic Data Types (Enum)

Lateralus v1.0 supports sum types through the enum declaration. Each variant is either a unit tag or a tuple-like constructor carrying one or more typed fields.

```
enum Shape {
    Circle(float),           # radius
    Rectangle(float, float), # width, height
    Triangle(float, float, float)
}

enum Tree {
```

```
    Leaf,
    Node(int, Tree, Tree)
}
```

Variants are constructed using the qualified form EnumName::Variant(args). Unit variants are referenced as EnumName::Variant without parentheses. The type of a unit variant is EnumName; the type of a constructor variant is a function type (T1, T2, ...) -> EnumName.

Enum types in v1.0 are monomorphic; the type parameters visible in the standard library (option[T], result[T, E]) are built-in generics handled by the compiler. User-defined generics arrive in v2.0.

The memory layout of an enum value is a tag word (8-bit in the interpreter) followed by a payload appropriate to the variant. Pattern matching (Section 14) deconstructs the tag and binds the payload fields.

14. Pattern Matching

Pattern matching is the primary mechanism for consuming enum values. The match expression evaluates a scrutinee, then tries each arm in order, binding the first matching pattern.

```
match_expr ::= 'match' expr '{' arm+ '}'
arm        ::= pattern '=>' (expr | block) ','
pattern    ::= '_'                    # wildcard
             | literal                # literal pattern
             | ident                  # binding pattern
             | EnumName '::' Variant '(' pattern_list? ')'
             | EnumName '::' Variant  # unit variant
```

```
fn area(s: Shape) -> float {
    match s {
        Shape::Circle(r)      => 3.14159 * r * r,
        Shape::Rectangle(w, h)=> w * h,
        Shape::Triangle(a, b, c) => {
            let p = (a + b + c) / 2.0;
            std::math::sqrt(p * (p-a) * (p-b) * (p-c))
        },
    }
}
```

Match arms are checked for exhaustiveness: the compiler MUST report a warning if the set of patterns does not cover all variants of the scrutinee type. In v1.0 exhaustiveness is checked only for enum scrutinees; match on int or str requires a wildcard or literal-covering arm.

Patterns bind new names in the arm body. A binding pattern (a bare identifier) matches any value and introduces the identifier as a local binding. The wildcard _ matches any value without introducing a binding. Guard expressions are not supported in v1.0; they arrive in v2.0.

All arms of a match expression MUST produce the same type. The type of the overall match expression is that common type. If arms produce incompatible types, the type checker reports a mismatch at the first diverging arm.

15. The Module System (v1.0)

In v1.0 the module system is rudimentary: each file is a module and modules are identified by their file path relative to the project root. There is no hierarchical namespace; all modules exist at one level.

```
import_decl ::= 'import' module_path ('use' ident_list)?
module_path ::= string_literal
```

```
ident_list  ::= ident (',' ident)*
```

An import makes the named module's pub-marked declarations available under the module's basename. For example, import "utils" makes utils::helper() available. The use clause selectively imports names into the current scope without qualification: import "utils" use helper makes helper available directly.

Circular imports are detected at load time and produce a fatal error. The import resolver performs a DFS; if a module is encountered while its own resolution is still in progress, the cycle is reported with the full chain.

The module system is substantially redesigned in v3.0. This section describes v1.0 only. Implementers building v1.0 tooling should treat the module system as minimal scaffolding rather than a production feature.

16. Standard Library (v1.0)

The v1.0 standard library is intentionally minimal. It is distributed as part of the interpreter and is always available without an explicit import.

16.1 std::io

```
# std::io
fn print(s: str) -> unit
fn println(s: str) -> unit
fn read_line() -> str
fn read_file(path: str) -> result[str, str]
fn write_file(path: str, content: str) -> result[unit, str]
```

16.2 std::string

```
# std::string
fn len(s: str) -> int
fn concat(a: str, b: str) -> str
fn slice(s: str, start: int, end: int) -> str
fn split(s: str, sep: str) -> list[str]
fn contains(s: str, sub: str) -> bool
fn parse_int(s: str) -> result[int, str]
fn parse_float(s: str) -> result[float, str]
fn to_upper(s: str) -> str
fn to_lower(s: str) -> str
fn trim(s: str) -> str
```

16.3 std::list

```
# std::list
fn map(f: (A) -> B, xs: list[A]) -> list[B]
fn filter(f: (A) -> bool, xs: list[A]) -> list[A]
fn fold(f: (B, A) -> B, init: B, xs: list[A]) -> B
fn len(xs: list[A]) -> int
fn append(xs: list[A], ys: list[A]) -> list[A]
fn head(xs: list[A]) -> option[A]
fn tail(xs: list[A]) -> list[A]
fn nth(xs: list[A], i: int) -> option[A]
fn sum(xs: list[int]) -> int
fn join(xs: list[str], sep: str) -> str
```

16.4 std::option and std::result

```
# std::option
fn is_some(o: option[A]) -> bool
fn is_none(o: option[A]) -> bool
fn unwrap(o: option[A]) -> A    # runtime error if None
fn unwrap_or(o: option[A], default: A) -> A
fn map_opt(f: (A) -> B, o: option[A]) -> option[B]

# std::result
fn is_ok(r: result[T, E]) -> bool
fn is_err(r: result[T, E]) -> bool
fn unwrap_ok(r: result[T, E]) -> T
fn unwrap_err(r: result[T, E]) -> E
fn map_ok(f: (T) -> U, r: result[T, E]) -> result[U, E]
```

17. Type Inference Rules

The following judgment forms define the static semantics of v1.0. Gamma |- e : t reads 'under environment Gamma, expression e has type t'.

```
-- Var
(x : t) in Gamma
----------------
Gamma |- x : t

-- Int literal
----------------
Gamma |- n : int

-- Application
Gamma |- e1 : (t1) -> t2    Gamma |- e2 : t1
-------------------------------------------
Gamma |- e1(e2) : t2

-- Pipeline
Gamma |- e1 : A    Gamma |- e2 : (A) -> B
----------------------------------------
Gamma |- e1 |> e2 : B

-- Let
Gamma |- e1 : t1    Gamma, x:generalize(Gamma, t1) |- e2 : t2
-------------------------------------------------------------
Gamma |- let x = e1 in e2 : t2
```

The function and pattern-match rules are omitted for space but follow standard HM conventions. The generalize function abstracts type variables free in t1 but not free in Gamma, producing a type scheme.

The error pipe rule adds a result-unwrapping step: if e1 : result[A, E] and e2 : (A) -> result[B, E], then e1 |?> e2 : result[B, E]. The error type E must unify across all |?> steps in a chain.

18. Operational Semantics (Selected Rules)

Operational semantics are given as a small-step reduction relation e --> e' on closed expressions. The full set of rules is 47; we present the most distinctive ones.

```
-- E-Pipe (|>)
v is a value    f is a function value
-------------------------------------
v |> f --> f(v)

-- E-PipeErr-Ok (|?>)
result::Ok(v) |?> f --> f(v)

-- E-PipeErr-Err (|?>)
result::Err(e) |?> f --> result::Err(e)

-- E-Match-Hit
pattern p matches value v with bindings sigma
---------------------------------------------
match v { p => e, ... } --> e[sigma]

-- E-Match-Miss
pattern p does not match value v
----------------------------------------
match v { p => e1, rest } --> match v { rest }
```

Congruence rules (reducing sub-expressions in context) are standard and not listed here. Values in v1.0 are: integer literals, float literals, boolean literals, string literals, unit (), list literals with value elements, enum constructors applied to value arguments, and closures.

19. Memory Model (Value Semantics)

The v1.0 memory model is value semantics throughout. When a value is passed to a function or bound with let, the receiving binding holds an independent copy of the value. Primitive types (int, float, bool, unit) are copied by value at the machine level.

String values are immutable in v1.0. The interpreter may share string data between bindings (interning), but no operation can observe sharing: the language model is always a fresh copy. Operations like std::string::concat always return new strings.

List values copy the spine (the array of element references) on each binding. Elements themselves are shared as heap references. Since no mutation of elements is exposed in v1.0, this sharing is unobservable and the model is still pure value semantics.

Enum values copy the tag and recursively copy all payload fields. For deeply nested tree-shaped data this copying has O(n) cost where n is the number of nodes. A persistent data structure library is planned for v2.0 to address this.

There is no explicit memory management in v1.0. The interpreter uses reference counting for heap objects (strings, lists, enum payloads). Cycles are not possible because v1.0 has no mutable references. The reference-counting cost is amortized into allocation and deallocation; it is not visible at the language level.

20. REPL Mode

The Lateralus v1.0 distribution includes an interactive read-eval-print loop. The REPL evaluates statements and expressions one at a time and prints the result after each successful evaluation.

```
$ lateralus
Lateralus v1.0 REPL — type :help for commands
```

```
ltl> let x = [1, 2, 3];
ltl> x |> std::list::map(fn(n: int) -> int { n * 2 })
[2, 4, 6]
ltl> :type x
list[int]
ltl> :quit
```

REPL commands are prefixed with : and are not valid Lateralus syntax. Available REPL commands in v1.0: :type expr (print inferred type), :load file (load a file into scope), :reset (clear the environment), :help, :quit.

The REPL environment persists across evaluations within a session. A binding introduced in one evaluation is available in all subsequent evaluations. Shadowing a previous REPL binding is allowed and works identically to shadowing in source files.

Error messages in the REPL include the source position within the submitted line and the inferred types of subexpressions when relevant. The REPL does not crash on error; it prints the error and waits for the next input.

21. Known Limitations in v1.0

The following limitations are known and are addressed in later versions:

- **No user-defined generics.** Only the built-in generic types (list, option, result) support type parameters.
- **No tail-call optimization.** Deep recursion will stack-overflow.
- **No mutable references.** Stateful algorithms require functional patterns or explicit accumulator passing.
- **Single-file module system.** Multi-file projects require manual concatenation or the rudimentary import mechanism.
- **No async or concurrency primitives.** The runtime is single-threaded.
- **No FFI.** Interoperability with C or Python libraries is not supported.
- **No type classes or traits.** Ad-hoc polymorphism is not expressible.
- **Integer overflow is undefined behavior** in the interpreter on platforms where the host Python interpreter wraps or truncates.

22. Versioning Policy

Lateralus follows semantic versioning (SemVer 2.0). A version number is MAJOR.MINOR.PATCH. The MAJOR version increments on breaking changes to the grammar, type system, or standard library. MINOR increments on new features that are backward-compatible. PATCH increments on bug fixes.

This document specifies v1.0.0. All v1.x.y versions MUST accept every program that v1.0.0 accepts and MUST assign it the same semantics. New features added in v1.x.y MUST NOT conflict with the grammar or semantics defined here.

v2.0 is a planned breaking release. Programs written for v1.0 may require syntactic updates when migrated to v2.0, particularly around the module system and the introduction of mutable bindings. A migration guide will accompany the v2.0 release.

Appendix A: Reserved Keywords

The following identifiers are reserved in v1.0 and may not be used as user-defined names:

```
fn       let      if       else     match    case     return
enum     import   pub      use      true     false    and
or       not      in       for      while    break    continue
unit     int      float    str      bool
```

The type names unit, int, float, str, and bool are reserved identifiers even though they appear in the type grammar position rather than the expression grammar position. Future versions may relax this restriction for type-level namespacing.

Appendix B: Operator Precedence Table

Operators are listed from highest to lowest binding power. All binary operators at the same level are left-associative.

```
Level  Operator(s)           Associativity
-----------------------------------------------
1      - (unary)  not        right
2      *  /  %               left
3      +  -                  left
4      ==  !=  <  <=  >  >= left
5      and                   left
6      or                    left
7      |>  |?>               left
```

Function application (call syntax) binds tighter than all binary operators and is evaluated left-to-right for chained calls. Grouping with parentheses overrides all precedence rules.