Lateralus Language Specification v3

Complete grammar, type system, pipeline semantics, and standard library surface

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT This document is the normative specification for Lateralus version 3. It supersedes the v1 and v2 specifications. Version 3 adds: the effect type system (§4), async pipelines with structured concurrency (§5), the formal module system (§8), and the polyglot bridge interface (§9). The grammar in §2 and the core type system in §3 are unchanged from v2 except for additions required by the new features. Implementations must conform to all normative sections; informative sections are labeled [INF].

1. Lexical Structure

Lateralus source files are UTF-8 encoded. The lexical grammar defines tokens in terms of Unicode categories.

```
-- Pipeline operators (longest-match rule)
PIPE_TOTAL   ::= '|>'
PIPE_FALLIBLE ::= '|?>'
PIPE_ASYNC   ::= '|>>'
PIPE_COLLECT ::= '|>|'

-- Identifiers
IDENT ::= (XID_Start | '_') XID_Continue*

-- Numeric literals
INT   ::= [0-9]+ | '0x' [0-9A-Fa-f]+ | '0b' [01]+
FLOAT ::= [0-9]+ '.' [0-9]+ ([eE] [+-]? [0-9]+)?
```

Comments: -- to end of line (single-line); --- lines are documentation comments attached to the following declaration. Block comments are not part of the grammar; multi-line comments must use repeated -- lines.

2. Grammar

Top-level productions:

```
program     ::= item*
item        ::= fn_def | record_def | enum_def | module_def
              | use_decl | impl_def | trait_def

fn_def      ::= doc_comment? 'fn' IDENT generics? params return_type? body
pipeline    ::= expr (pipe_op expr)*
pipe_op     ::= PIPE_TOTAL | PIPE_FALLIBLE | PIPE_ASYNC | PIPE_COLLECT

expr        ::= pipeline | match_expr | block | literal | IDENT
              | fn_call | field_access | lambda | if_expr
```

The pipeline production is left-associative. Operator precedence: pipeline operators bind more loosely than function application but more tightly than let and if expressions.

3. Core Type System

Lateralus uses a Hindley-Milner type system extended with row polymorphism for records and effect typing (§4). Type inference is complete for the HM fragment; effect annotations may be required at function boundaries.

```
Type τ ::= Int | Float | Bool | Str | Unit
         | τ -> τ                    -- function
```

```
        | Result<τ, τ>              -- fallible
        | Async<τ>                  -- async value
        | Vec<τ> | Option<τ>        -- containers
        | {l: τ, ...| ρ}            -- open record
        | α                         -- type variable
        | ∀α. τ                     -- polymorphic
```

Pipeline operator typing rules (normative):

```
Γ ■ e : A    Γ ■ f : A → B
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  [PIPE-TOTAL]
Γ ■ e |> f : B

Γ ■ e : Result<A,E>    Γ ■ f : A → Result<B,E>
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  [PIPE-FALLIBLE]
Γ ■ e |?> f : Result<B,E>
```

4. Effect Type System

Version 3 adds a row-polymorphic effect type system. Effects are tracked in the return type of functions:

```
-- Effect rows
Effect ε ::= {}          -- pure
           | {IO | ε}    -- I/O
           | {Exc E | ε} -- exception of type E
           | {State S | ε} -- mutable state of type S
           | ε           -- row variable

-- Functions are typed with their effect row
fn read_file(path: Str) -> Str ! {IO}
fn pure_map<A,B>(f: A->B, xs: Vec<A>) -> Vec<B> ! {}
```

Effect polymorphism allows higher-order functions to propagate effects from their arguments without annotation:

```
fn map<A,B,ε>(f: A -> B ! ε, xs: Vec<A>) -> Vec<B> ! ε
-- ε is inferred from the argument f at call sites
```

5. Async Pipelines and Structured Concurrency

The |>> operator applies a function to each element of a stream concurrently. Structured concurrency guarantees that all concurrent tasks are joined before the pipeline stage completes:

```
-- Async pipeline semantics
Γ ■ e : Stream<A>    Γ ■ f : A → Async<B>
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  [PIPE-ASYNC]
Γ ■ e |>> f : Stream<B>

-- Collect terminates a stream into a Vec
Γ ■ e : Stream<A>
■■■■■■■■■■■■■■■■■■■■  [PIPE-COLLECT]
Γ ■ e |>| : Vec<A>
```

All tasks spawned by |>> are owned by the pipeline stage's nursery. If any task fails, the nursery cancels all sibling tasks and propagates the failure. Dangling tasks are impossible.

6. Pattern Matching

Pattern matching is exhaustive: the compiler rejects a match expression that does not cover all constructors of the matched type. Guards are permitted but do not contribute to exhaustiveness:

```
match value {
    Ok(x) if x > 0  => positive(x),
    Ok(x)           => non_positive(x),
    Err(e)          => handle_error(e),
    -- No wildcard needed: Ok and Err exhaust Result<A,E>
}
```

Structural patterns, tuple patterns, range patterns (1..=10), and or-patterns (A | B) are all supported. Binding patterns (x @ pattern) bind the matched value while also matching its structure.

7. Ownership and Lifetimes

Lateralus uses a single-owner memory model. Every value has exactly one owner; ownership is transferred (moved) on assignment and function call. Borrows are immutable (&T;) or mutable (&mut; T); at most one mutable borrow may exist at a time:

```
fn process(data: Vec<u8>) -> usize {    -- data is moved in
    let len = data.len();               -- immutable borrow
    transform(&mut data);               -- mutable borrow
    len                                 -- data freed here
}

-- Lifetime annotations for struct fields containing borrows
record View<'a> { data: &'a [u8] }
```

The borrow checker runs after type inference and before code generation. Its output is a set of lifetime constraints; violations are reported as errors with suggested fixes.

8. Module System and Packages

Lateralus has a two-level namespace: modules (within a package) and packages (in the registry). The module path syntax is package::module::item:

```
-- Declare a module
module crypto {
    pub fn sha256(data: &[u8]) -> [u8; 32] { ... }
    -- private by default
    fn compress(state: &mut State, block: &[u8]) { ... }
}

-- Use from another module
use crypto::sha256;
use crypto::{sha256, hmac_sha256};
```

Visibility modifiers: pub (public), pub(package) (visible within the package), and private (default). There is no pub(super); packages are the visibility boundary.

9. Polyglot Bridge

The polyglot bridge allows Lateralus code to call C, Rust, and Python functions and vice versa. Bindings are declared with extern blocks:

```
-- Call a C function
extern "C" {
    fn strlen(s: *const u8) -> usize;
    fn malloc(size: usize) -> *mut u8;
}

-- Export a Lateralus function to C
#[export_c]
pub fn add(a: i32, b: i32) -> i32 { a + b }
```

All calls across the bridge are unsafe: the Lateralus type system cannot verify the safety of foreign code. The bridge provides automatic type marshalling for primitive types; complex types require explicit marshalling code.