The Lateralus Programming Language Specification v1.0

bad-antics | September 2024 | Language Specification

Abstract

This document is the formal specification of the Lateralus programming language version 1.0. It defines the complete syntax, semantics, type system, ownership model, pattern matching, module system, and standard library interfaces for conforming implementations of the language.

1 Introduction

This document is the formal specification of the Lateralus programming language version 1.0. Lateralus is a systems programming language featuring pipeline-native data flow, ownership-based memory safety, algebraic data types, and pattern matching.

The specification defines the syntax, semantics, type system, and standard library interfaces for conforming implementations. Version 1.0 represents the first stable release of the language.

Lateralus targets bare-metal, operating system kernel, and application development. The language compiles to native code via RISC-V and x86-64 backends, and to bytecode for the Lateralus Virtual Machine.

2 Lexical Structure

A Lateralus source file is a sequence of Unicode characters encoded in UTF-8. The lexer transforms the character stream into a sequence of tokens. Whitespace and comments are not significant except as token separators.

Identifiers begin with a letter or underscore, followed by letters, digits, or underscores. Unicode letters from all scripts are permitted. Identifiers are normalized to NFC before comparison.

Keywords: pipe, fn, let, mut, if, else, match, struct, enum, return, loop, while, for, in, break, continue, use, pub, mod, trait, impl, self, Self, true, false, as, ref, move, type, where, const, static, unsafe, extern, async, await.

Numeric literals: decimal (42), hexadecimal (0xFF), binary (0b1010), octal (0o17), floating-point (3.14, 1e-5). Underscores may appear between digits for readability (1_000_000).

String literals: double-quoted strings with escape sequences. Raw strings (r-prefix) disable escape processing. Multi-line strings use triple quotes. Interpolated strings use curly braces for expressions.

3 Types

Primitive types: i8, i16, i32, i64, u8, u16, u32, u64, f32, f64, bool, char, unit. Integer types are signed

or unsigned. Floating-point types follow IEEE 754. The unit type has a single value ().

Tuple types: (T1, T2, ..., Tn) is a tuple of n elements. Tuple elements are accessed by index: t.0, t.1, etc. The empty tuple () is the unit type.

Array types: [T; N] is an array of N elements of type T. Array length is part of the type. Slice types [T] represent dynamically-sized views into arrays.

Reference types: &T is an immutable reference to T. &mut T is a mutable reference to T. References are non-null pointers with borrow-checked lifetimes.

Function types: fn(T1, T2) -> R is a function taking T1 and T2 and returning R. Function types are first-class values.

Struct types are nominal: two structs with identical fields but different names are distinct types. Structs are defined with the struct keyword.

Enum types define a closed set of variants. Each variant can carry data. Enums are matched with the match expression.

4 Expressions

Literal expressions: integer literals, floating-point literals, string literals, boolean literals (true, false), and the unit literal ().

Arithmetic expressions: addition (+), subtraction (-), multiplication (*), division (/), remainder (%). Arithmetic operators work on numeric types. Mixed-type arithmetic requires explicit conversion.

Comparison expressions: equal (==), not equal (!=), less than (<), greater than (>), less or equal (<=), greater or equal (>=). Comparisons return bool.

Logical expressions: and (&&), or (||), not (!). Logical operators work on bool. && and || use short-circuit evaluation.

Pipeline expressions: expr |> func chains data through transformations. The left operand is passed as the first argument to the right operand. Pipelines are left-associative.

Block expressions: { stmt; stmt; expr } evaluates statements in sequence and returns the final expression. Blocks introduce a new scope.

If expressions: if condition { then_expr } else { else_expr }. Both branches must have the same type. The else branch is optional when the then branch has type unit.

Match expressions: match expr { pattern => body, ... } matches the scrutinee against patterns in order. The first matching pattern's body is evaluated. Match must be exhaustive.

5 Statements

Let bindings: let x = expr; introduces a new variable. let mut x = expr; introduces a mutable variable. Type annotations are optional: let x: T = expr;

Assignment: x = expr; assigns a new value to a mutable variable. Compound assignment: x += expr; x -= expr; x *= expr; x /= expr; x %= expr;

Expression statements: expr; evaluates the expression for its side effects. The result is discarded.

Return statements: return expr; returns from the current function. The expression's type must match the function's return type.

6 Items

Function definitions: fn name(param: Type, ...) -> ReturnType { body }. Functions are items that can be called. Functions may be generic: fn name[T](param: T) -> T.

Struct definitions: struct Name { field: Type, ... }. Structs define product types. Tuple structs: struct Name(Type, Type). Unit structs: struct Name;

Enum definitions: enum Name { Variant1, Variant2(Type), Variant3 { field: Type } }. Enums define sum types with named variants.

Trait definitions: trait Name { fn method(&self) -> Type; }. Traits define interfaces. Types implement traits with impl blocks: impl TraitName for TypeName { fn method(&self) -> Type { body } }.

Impl blocks: impl TypeName { fn method(&self) { body } }. Impl blocks add methods to types. Methods take self, &self, or &mut self as the first parameter.

Use declarations: use module::item; brings an item into scope. use module::*; imports all public items. use module::item as alias; creates an alias.

7 Ownership and Borrowing

Every value has exactly one owner. When the owner goes out of scope, the value is dropped. Ownership is transferred by assignment, function call, and return.

Immutable borrowing: &expr creates an immutable reference. Multiple immutable references can coexist. The value cannot be moved or mutated while immutably borrowed.

Mutable borrowing: &mut expr creates a mutable reference. Only one mutable reference can exist at a time. No other references (mutable or immutable) can coexist.

Lifetimes: references have lifetimes that ensure the referenced value outlives the reference. Lifetimes are usually inferred. Explicit lifetime annotations: &'a T.

Move semantics: assignment and function call move the value by default. After a move, the original variable is no longer usable. Copy types (integers, bools) are copied instead of moved.

8 Pattern Matching

Literal patterns: 42, 'a', true match the specific value. Variable patterns: x binds the matched value. Wildcard pattern: _ matches anything and discards the value.

Tuple patterns: (a, b, c) destructures tuples. Struct patterns: Name { field: pattern } destructures structs. Enum patterns: Variant(pattern) destructures enum variants.

Guard patterns: pattern if condition refines a pattern with a boolean condition. Or patterns: p1 | p2 matches if either pattern matches.

Exhaustiveness checking: the compiler verifies that match expressions cover all possible values. Missing patterns are compile-time errors. Wildcard arms satisfy exhaustiveness.

9 Modules and Visibility

Modules organize code into namespaces. Each file is a module. Subdirectories with mod.lat files define submodules. The module tree mirrors the directory structure.

Visibility: items are private by default. The pub keyword makes items visible to other modules. pub(crate) restricts visibility to the current crate.

Module paths: module::submodule::item refers to an item through the module hierarchy. Self refers to the current module. super refers to the parent module.

10 Conclusion

The Lateralus 1.0 specification defines a systems programming language that combines safety, performance, and expressiveness. Pipeline-native data flow, ownership-based memory safety, and algebraic data types provide a foundation for safe and efficient systems programming.

2.1 Operators and Punctuation

Arithmetic operators: + (addition), - (subtraction/negation), * (multiplication), / (division), % (remainder). Operator precedence follows mathematical convention.

Comparison operators: == (equality), != (inequality), < (less than), > (greater than), <= (less or equal), >= (greater or equal). Comparison operators return bool.

Logical operators: && (short-circuit and), || (short-circuit or), ! (not). Logical operators work on bool values only.

Bitwise operators: & (and), | (or), ^ (xor), ! (not), << (left shift), >> (right shift). Bitwise operators work on integer types.

Assignment operators: = (assign), += (add-assign), -= (sub-assign), *= (mul-assign), /= (div-assign), %= (rem-assign), &= (and-assign), |= (or-assign), ^= (xor-assign), <<= (shl-assign), >>= (shr-assign).

Pipeline operator: |> passes the left operand as the first argument to the right operand. Pipeline expressions are left-associative. The pipeline operator has lower precedence than arithmetic.

Range operators: .. (exclusive range), ..= (inclusive range). Ranges are used in for loops and array slicing. Ranges are first-class values of type Range[T].

Punctuation: ( ) (grouping/tuples), [ ] (arrays/indexing), { } (blocks/structs), , (separator), ; (statement terminator), : (type annotation), :: (path separator), . (field access), -> (return type), => (match arm), # (attribute).

Operator precedence (highest to lowest): unary (- !), multiplicative (* / %), additive (+ -), shift (<< >>), bitwise and (&), bitwise xor (^), bitwise or (|), comparison (== != < > <= >=), logical and (&&), logical or (||), pipeline (|>), assignment (= += etc).

Operator associativity: binary arithmetic operators are left-associative. Assignment operators are right-associative. The pipeline operator is left-associative. Comparison operators are non-associative (cannot chain).

3.1 Type Inference

Lateralus uses bidirectional type inference based on Hindley-Milner type inference with extensions for ownership and lifetimes.

Local type inference: variable types are inferred from their initializers. let x = 42; infers x: i32. let s = 'hello'; infers s: &str.

Function return type inference: the return type can be omitted when it can be inferred from the function body. fn double(x: i32) { x * 2 } infers return type i32.

Generic type inference: generic type parameters are inferred from arguments. let v = Vec::new(); v.push(42); infers Vec[i32].

Integer literal inference: unadorned integer literals (42) default to i32. When context requires a different type, the literal is inferred accordingly: let x: u64 = 42;

Float literal inference: unadorned float literals (3.14) default to f64. Context-dependent inference selects f32 when required.

Closure type inference: closure parameter types and return types are inferred from usage context. let f = |x| x + 1; infers the closure type from how f is called.

Type inference limitations: function parameter types must be annotated. Struct field types must be annotated. Generic type parameters on items must be declared.

Error recovery in type inference: when inference fails, the compiler assigns an error type to the expression. Subsequent errors involving the error type are suppressed.

Type inference performance: inference runs in O(n * alpha(n)) time where n is the number of expressions and alpha is the inverse Ackermann function (effectively constant).

4.1 Pipeline Expressions Detailed

Simple pipeline: expr |> f passes the value of expr as the first argument to function f. The result of f becomes the pipeline value.

Chained pipeline: expr |> f |> g |> h is equivalent to h(g(f(expr))). Each stage receives the output of the previous stage.

Pipeline with additional arguments: expr |> f(a, b) is equivalent to f(expr, a, b). The pipeline value is inserted as the first argument.

Pipeline with placeholder: expr |> f(a, _, b) is equivalent to f(a, expr, b). The underscore _ marks where the pipeline value is inserted.

Pipeline with closures: expr |> |x| x + 1 applies an inline closure as a pipeline stage. The closure receives the pipeline value as its parameter.

Pipeline with method syntax: expr |> .method() is equivalent to expr.method(). The dot prefix indicates a method call on the pipeline value.

Pipeline error handling: expr |> f? |> g? uses the ? operator to propagate errors. If f returns Err, the pipeline short-circuits and returns the error.

Pipeline type checking: each stage's output type must match the next stage's input type. Type mismatches are reported with the stage number and the mismatched types.

Pipeline optimization: the compiler fuses consecutive map operations, eliminates identity stages, and specializes generic stages. Optimized pipelines have zero overhead compared to manual function calls.

Pipeline semantics: pipeline evaluation is strict (eager) by default. Lazy pipelines are available through the lazy keyword: expr |> lazy f |> lazy g evaluates only when the result is consumed.

5.1 Control Flow Statements

While loops: while condition { body } evaluates body repeatedly while condition is true. The condition is evaluated before each iteration. Break exits the loop. Continue skips to the next iteration.

For loops: for pattern in iterable { body } iterates over elements of an iterable. The pattern binds each element. The iterable must implement the Iterator trait.

Loop expressions: loop { body } repeats indefinitely. Break with a value exits the loop and returns the value: let x = loop { if done { break result; } }.

If-let: if let pattern = expr { body } matches the expression against the pattern. The body executes only if the pattern matches. Useful for Option and Result types.

While-let: while let pattern = expr { body } combines while and pattern matching. The loop continues while the pattern matches the expression.

For-in with ranges: for i in 0..10 { body } iterates over integers 0 through 9. for i in 0..=10 includes 10. Ranges are lazy iterators.

Break with labels: 'outer: loop { 'inner: loop { break 'outer; } } breaks from a named loop. Labels begin with a single quote.

Loop else: for x in iter { body } else { no_match_body } executes the else block if the loop completes without break. Useful for search patterns.

Loop invariant assertions: debug_assert in loop bodies are checked on every iteration in debug mode. Loop invariants help verify correctness during development.

Loop optimization hints: the #[unroll] attribute hints the compiler to unroll small loops. The #[vectorize] attribute hints the compiler to use SIMD instructions.

6.1 Generic Items

Generic functions: fn identity[T](x: T) -> T { x } defines a function generic over type T. The compiler monomorphizes generic functions for each concrete type used.

Generic structs: struct Pair[A, B] { first: A, second: B } defines a struct generic over types A and B. Concrete types are specified at use: Pair[i32, String].

Generic enums: enum Option[T] { Some(T), None } defines an enum generic over T. Generic enums are commonly used for: Option (optional values), Result (error handling), and collections.

Generic impl blocks: impl[T] Vec[T] { fn push(&mut self, value: T) { ... } } implements methods for all Vec types. Trait bounds constrain T: impl[T: Display] Vec[T] { fn print(&self) { ... } }.

Trait bounds: fn print[T: Display](x: T) constrains T to implement Display. Multiple bounds: T: Display + Clone. Where clauses: fn f[T](x: T) where T: Display + Clone.

Associated types: trait Iterator { type Item; fn next(&mut self) -> Option[Self::Item]; } declares an associated type. Implementors specify the concrete type.

Generic constants: const MAX[T: Bounded]: T = T::max_value(); defines a constant generic over T. The value is computed at compile time for each concrete type.

Generic type aliases: type Pair[T] = (T, T); creates a generic type alias. Type aliases do not create new types; they are abbreviations.

Monomorphization: the compiler generates specialized code for each concrete type. Vec[i32] and Vec[String] produce separate compiled functions. Monomorphization enables zero-cost generics.

Generic type bounds checking: the compiler verifies that generic code only uses operations available on the bounded type. Using an operation not provided by the bounds is a compile error.

7.1 Ownership Rules Formal

Rule 1: Each value has exactly one owner at any time. The owner is the variable that holds the value. When the owner goes out of scope, the value is dropped.

Rule 2: Ownership is transferred by: assignment (let y = x;), function argument (f(x)), and return value (return x). After transfer, the original owner is invalidated.

Rule 3: Copy types are duplicated instead of moved. A type is Copy if all its fields are Copy. Primitive types (integers, floats, bools, chars) are Copy.

Rule 4: At most one mutable reference (&mut T) to a value can exist at a time. While a mutable reference exists, no other references to the value can exist.

Rule 5: Any number of immutable references (&T) to a value can coexist. While immutable references exist, the value cannot be moved or mutated.

Rule 6: References must not outlive the value they reference. The compiler checks that the reference's lifetime is contained within the value's lifetime.

Rule 7: Struct fields can be individually borrowed. Borrowing field A does not prevent borrowing field B. The compiler tracks per-field borrow states.

Rule 8: Values in collections are owned by the collection. Borrowing a collection element borrows from the collection. Mutable iteration requires mutable borrowing.

Rule 9: Closures that capture by reference borrow the captured variables. The closure cannot outlive the captured variables. Move closures take ownership of captured variables.

Rule 10: Unsafe blocks disable ownership checking for raw pointer operations. Unsafe code is the programmer's responsibility. Safe wrappers encapsulate unsafe operations.

8.1 Pattern Matching Formal

Pattern syntax: Pattern = LiteralPat | IdentPat | WildcardPat | TuplePat | StructPat | EnumPat | RefPat | GuardPat | OrPat | RangePat.

Literal patterns match specific values. Integer literal patterns: 42 matches the integer 42. String literal patterns: 'hello' matches the string 'hello'. Boolean: true, false.

Identifier patterns bind the matched value to a variable. The variable is available in the arm's body. Identifier patterns always match (they are irrefutable).

Wildcard pattern (_) matches any value and discards it. Wildcard is irrefutable. Used for: ignoring values in tuples, providing catch-all match arms, and ignoring function results.

Tuple patterns (a, b, c) destructure tuples. The number of elements must match. Nested patterns are allowed: (a, (b, c)).

Struct patterns Name { field: pattern, .. } destructure structs. The .. syntax ignores unmentioned fields. Field shorthand: Name { x, y } binds x and y from fields named x and y.

Enum patterns Variant(pattern) destructure enum variants. Nested patterns allow deep matching: Some(Ok(value)).

Reference patterns &pattern and &mut pattern match references. The inner pattern matches the referenced value. Reference patterns are used in match on borrowed values.

Range patterns: 0..=9 matches integers in the range. 'a'..='z' matches characters. Ranges must be non-empty. Range patterns are used for: ASCII classification and numeric ranges.

Guard patterns: pattern if condition refines the match. The guard expression can reference variables bound by the pattern. Guards are evaluated after pattern matching.

9.1 Module System Details

Crate: the top-level compilation unit. A crate contains a module tree. The crate root is src/main.lat (for executables) or src/lib.lat (for libraries).

Module declaration: mod name; declares a submodule. The compiler searches for: name.lat in the current directory, or name/mod.lat in a subdirectory.

Inline modules: mod name { items } defines a module inline. Inline modules are useful for organizing code within a single file.

Module re-export: pub use module::item; re-exports an item from a submodule. Re-exports create public API surfaces without exposing internal module structure.

Prelude: the standard library prelude is automatically imported into every module. The prelude contains: Option, Result, Vec, String, common traits (Clone, Debug, Display, Iterator).

External crates: extern crate name; imports an external crate. In practice, dependencies are declared in the build manifest and automatically available.

Module privacy: private items are accessible only within the defining module and its submodules. Public items are accessible from any module.

Path resolution: paths are resolved relative to the current module. Absolute paths start with crate::. Relative paths start with self:: or super::. External paths start with the crate name.

Module initialization: modules are initialized in dependency order. Constants and statics are initialized at program startup. Circular initialization is detected and reported.

Conditional compilation: #[cfg(feature = 'name')] conditionally includes items based on compilation features. Features are declared in the build manifest.

3.2 Pointer Types

Raw pointers: *const T (immutable raw pointer) and *mut T (mutable raw pointer). Raw pointers can be null. Dereferencing raw pointers requires an unsafe block.

Box type: Box[T] is a heap-allocated value. Box provides owned heap allocation. When the Box goes out of scope, the heap memory is freed.

Rc type: Rc[T] is a reference-counted pointer. Multiple Rc values can share ownership. The value is dropped when the last Rc is dropped. Rc is not thread-safe.

Arc type: Arc[T] is an atomically reference-counted pointer. Arc is thread-safe. Arc uses atomic operations for the reference count, which is slower than Rc.

Weak type: Weak[T] is a weak reference to an Rc or Arc value. Weak references do not prevent deallocation. Upgrading a Weak to an Rc/Arc returns None if the value has been dropped.

Cell type: Cell[T] provides interior mutability for Copy types. Cell allows mutation through shared references. Cell uses move semantics: get returns a copy, set replaces the value.

RefCell type: RefCell[T] provides interior mutability with runtime borrow checking. borrow() returns a shared reference. borrow_mut() returns an exclusive reference. Violations panic.

Slice type: &[T] is a dynamically-sized view into a contiguous sequence. Slices consist of: a pointer to the first element and a length. Slices are always borrowed.

String types: String is an owned, heap-allocated, growable UTF-8 string. &str is a borrowed string slice. String literals have type &str with static lifetime.

Function pointers: fn(T) -> R is a function pointer type. Function pointers can point to any function with the matching signature. Function pointers are Copy.

4.2 Closure Expressions

Closure syntax: |params| body creates a closure. Parameter types are inferred: |x| x + 1 infers the type from context. Explicit types: |x: i32| -> i32 { x + 1 }.

Capture modes: closures capture variables from their environment. By default, the closure borrows variables. The move keyword forces move capture: move |x| x.

Closure traits: closures implement one or more of: Fn (callable through shared reference), FnMut (callable through mutable reference), FnOnce (callable by consuming the closure).

Closure types: each closure has a unique anonymous type. Closures with the same signature but different captures have different types. Trait objects (dyn Fn) erase the concrete type.

Closure coercion: non-capturing closures can be coerced to function pointers. |x: i32| x + 1 can be used where fn(i32) -> i32 is expected.

Closure size: closures that capture by reference are 8 bytes per captured reference. Move closures contain the captured values inline. Zero-capture closures are zero-sized.

Returning closures: functions that return closures use -> impl Fn(T) -> R. The concrete closure type is inferred by the compiler. Boxing is needed for dynamic dispatch: Box[dyn Fn].

Closure recursion: closures cannot directly call themselves. Recursive closures use a helper function or a fixed-point combinator.

Closure and ownership: closures that capture mutable references require the closure variable to be declared mut. The borrow checker verifies closure captures.

Closure performance: inline closures (known type) have zero overhead compared to function calls. Trait object closures (dynamic dispatch) add one indirect call (approximately 5 ns).

6.2 Trait Definitions Detailed

Trait syntax: trait Name { fn method(&self) -> Type; } declares a trait with a required method. Traits may have: required methods (no body), provided methods (with default body), and associated types.

Trait implementation: impl TraitName for TypeName { fn method(&self) -> Type { body } }. The implementation must provide all required methods. Provided methods can be overridden.

Supertraits: trait Child: Parent { } requires that implementing Child also implements Parent. Supertrait methods are available on Child trait objects.

Trait objects: dyn TraitName is a dynamically-dispatched trait type. Trait objects are fat pointers (data + vtable). Trait objects enable heterogeneous collections.

Object safety: a trait is object-safe if all methods: take self by reference, have no generic parameters, and return a type that does not mention Self (except as the receiver).

Blanket implementations: impl[T: Display] ToString for T { } provides ToString for all types implementing Display. Blanket impls reduce boilerplate.

Coherence: each trait implementation must be in the module that defines either the trait or the type. Coherence prevents conflicting implementations from different modules.

Sealed traits: a trait defined in a private module cannot be implemented outside the crate. Sealed traits enable: exhaustive matching on implementors and stable API evolution.

Marker traits: traits with no methods serve as compile-time markers. Common markers: Copy (value can be bitwise copied), Send (value can be sent to another thread), Sync (value can be shared between threads).

Trait aliases: trait ReadWrite = Read + Write; creates an alias for a combination of traits. Trait aliases simplify complex bounds.

7.2 Lifetime Annotations

Lifetime syntax: 'a is a lifetime parameter. Lifetimes are declared on functions, structs, and impl blocks. Lifetimes constrain how long references are valid.

Function lifetimes: fn longest['a](x: &'a str, y: &'a str) -> &'a str declares that the return reference lives as long as both inputs.

Struct lifetimes: struct Ref['a] { data: &'a str } declares that the struct borrows data with lifetime 'a. The struct cannot outlive the borrowed data.

Lifetime elision: the compiler infers lifetimes in common patterns. Rule 1: each reference parameter gets a distinct lifetime. Rule 2: single input lifetime is assigned to all outputs. Rule 3: &self lifetime is assigned to outputs.

Static lifetime: 'static is the longest lifetime. String literals have lifetime 'static. Global constants have lifetime 'static. 'static values live for the entire program.

Lifetime bounds: T: 'a means T's references are valid for at least lifetime 'a. 'a: 'b means lifetime 'a outlives lifetime 'b.

Lifetime in closures: closures that capture references inherit the lifetimes of the captured references. The closure cannot outlive any captured reference.

Higher-ranked lifetimes: for['a] fn(&'a T) -> &'a T represents a function that works for any lifetime. Higher-ranked lifetimes are used in trait bounds on closures.

Lifetime variance: &'a T is covariant in 'a (longer lifetime can be used for shorter). &'a mut T is invariant in 'a (exact lifetime required).

Lifetime debugging: the compiler can display inferred lifetimes with --explain-lifetimes. The output shows: each reference's lifetime, constraints, and the constraint solution.

2.2 Comments and Documentation

Line comments: // to end of line. Block comments: /* to */. Block comments nest: /* /* inner */ outer */ is a valid comment.

Documentation comments: /// for item documentation. //! for module documentation. Documentation comments support Markdown formatting.

Documentation code examples: code blocks in documentation comments are compiled and tested. This ensures that documentation examples stay in sync with the code.

Attributes: #[attribute] annotates the following item. Common attributes: #[derive(Debug, Clone)], #[test], #[cfg(condition)], #[inline], #[deprecated].

Inner attributes: #![attribute] annotates the enclosing item (usually the module). #![allow(unused)] suppresses unused variable warnings for the entire module.

Conditional compilation attributes: #[cfg(target_os = 'linux')], #[cfg(feature = 'async')], #[cfg(debug_assertions)]. cfg conditions can be combined with all, any, and not.

Deprecated items: #[deprecated(since = '1.5', note = 'Use new_function instead')] marks an item as deprecated. Using deprecated items generates warnings.

Inline attributes: #[inline] suggests inlining. #[inline(always)] forces inlining. #[inline(never)] prevents

inlining. Inlining decisions affect: performance, code size, and compilation time.

Test attributes: #[test] marks a function as a test. #[ignore] skips a test. #[should_panic] expects the test to panic. Test functions take no parameters and return () or Result.

Representation attributes: #[repr(C)] specifies C-compatible layout. #[repr(packed)] removes padding. #[repr(align(N))] specifies minimum alignment. Repr attributes control memory layout.

5.2 Error Handling

Result type: Result[T, E] represents success (Ok(T)) or failure (Err(E)). Functions that can fail return Result. Callers must handle both cases.

The ? operator: expr? unwraps Ok values and propagates Err values. In a function returning Result, expr? returns Err(e) if expr is Err(e). The ? operator chains error propagation.

Option type: Option[T] represents Some(T) or None. Option is used for values that may be absent. unwrap() extracts the value or panics if None.

Error conversion: the ? operator converts error types using the From trait. If f() returns Result[T, E1] and the function returns Result[U, E2], ? works when E2: From[E1].

Custom error types: enum MyError { IoError(io::Error), ParseError(String) }. Implementing From for standard error types enables ? operator usage.

Error trait: trait Error { fn description(&self) -> &str; fn cause(&self) -> Option[&Error]; }. The Error trait provides a common interface for error types.

Panic: panic('message') aborts the current thread with a message. Panics are for unrecoverable errors: logic bugs, out-of-bounds access, and assertion failures.

Catch unwind: catch_unwind(|| { code }) catches panics and returns Result. Used for: FFI boundaries, thread pools, and testing. Not all panics are catchable.

Error context: .context('message') adds context to an error. Error contexts create a chain of error messages. Display shows the full chain.

Result combinators: map, and_then, or_else, unwrap_or, unwrap_or_else. Combinators enable functional error handling without explicit match.

3.3 Numeric Type Semantics

Integer overflow: in debug mode, integer overflow panics. In release mode, integers wrap (two's complement). Explicit wrapping: x.wrapping_add(y). Checked: x.checked_add(y) returns Option.

Integer division: integer division truncates toward zero. 7 / 2 == 3. -7 / 2 == -3. Division by zero panics in both debug and release modes.

Floating-point semantics: follows IEEE 754-2019. NaN != NaN. Positive and negative zero are equal

(0.0 == -0.0). Infinity arithmetic follows IEEE rules.

Numeric conversions: widening conversions are implicit (i32 to i64). Narrowing conversions require as: let x: i32 = y as i32. Narrowing may truncate or change sign.

Numeric traits: Add, Sub, Mul, Div, Rem define arithmetic operators. Neg defines unary negation. PartialOrd and Ord define ordering. Numeric types implement these traits.

Saturating arithmetic: saturating_add, saturating_sub clamp to type bounds instead of wrapping. Used for: audio processing, image processing, and embedded systems.

Checked arithmetic: checked_add returns Some(result) or None on overflow. Checked operations are useful for: input validation and safe integer parsing.

Arbitrary precision: the standard library provides BigInt and BigRational for arbitrary-precision arithmetic. These types are heap-allocated and slower than fixed-size types.

SIMD types: the standard library provides: i32x4, f32x4, i64x2, f64x2 for SIMD operations. SIMD types support element-wise arithmetic, comparison, and shuffling.

Numeric literal suffixes: 42i32, 42u64, 3.14f32 specify the type of numeric literals. Suffixed literals do not require inference. Unsuffixed literals default to i32 or f64.

8.2 Exhaustiveness Algorithm

The exhaustiveness algorithm verifies that match expressions cover all possible input values. The algorithm constructs a matrix of patterns and checks for uncovered combinations.

Usefulness: a pattern is useful if there exists a value that matches the pattern but no previous pattern. A match is exhaustive if no additional useful pattern exists.

Algorithm U: the usefulness algorithm processes the pattern matrix column by column. For each column, it splits on the constructors present and recurses on the sub-patterns.

Constructor splitting: for enum types, constructors are the variants. For integer types, constructors are individual values and ranges. For tuples, the constructor is the tuple itself.

Wildcard expansion: the wildcard pattern _ is expanded to cover all constructors not covered by other patterns. This expansion enables recursive analysis.

Missing pattern reporting: when the match is not exhaustive, the algorithm constructs a witness pattern that represents an uncovered value. The witness is shown in the error message.

Redundant pattern detection: a pattern is redundant if it is not useful (no value matches it that does not match a previous pattern). Redundant patterns generate warnings.

Guard interaction: guards make exhaustiveness undecidable. The algorithm treats guarded patterns as partial: they match the pattern but may not execute the body. A wildcard arm is recommended.

Performance: the exhaustiveness algorithm has exponential worst-case complexity (product of

variant counts). In practice, match expressions are small and the algorithm runs in microseconds.

Nested pattern exhaustiveness: patterns with nested enum matching are handled by recursively checking each level. (Some(Ok(x)), None) is analyzed as: outer tuple, Option variants, Result variants.

9.2 Standard Library Overview

Core module: provides fundamental types and traits. Option, Result, iterators, basic traits (Clone, Debug, Display, Default, PartialEq, Eq, Hash, PartialOrd, Ord).

Collections module: Vec (growable array), HashMap (hash table), HashSet (hash set), BTreeMap (ordered map), BTreeSet (ordered set), VecDeque (double-ended queue), LinkedList.

I/O module: File, BufReader, BufWriter, stdin, stdout, stderr. Read and Write traits define the I/O interface. Path and PathBuf for file system paths.

String module: String (owned), str (borrowed slice). Methods: push, push_str, contains, replace, split, trim, to_uppercase, to_lowercase, parse.

Concurrency module: Thread (thread creation), Mutex (mutual exclusion), RwLock (read-write lock), Arc (atomic reference counting), Channel (message passing), Barrier.

Networking module: TcpStream, TcpListener, UdpSocket. Async networking: AsyncTcpStream, AsyncTcpListener. HTTP client and server through the http crate.

Time module: Instant (monotonic clock), SystemTime (wall clock), Duration (time span). Timer (one-shot or periodic). Sleep (thread sleep). Timeout (operation timeout).

Math module: trigonometric functions, logarithms, exponentials, rounding, min/max, clamping. Constants: PI, E, INFINITY, NAN.

Random module: thread-local random number generator. Distributions: uniform, normal, exponential. Seeded generators for reproducibility.

Testing module: assert!, assert_eq!, assert_ne! macros. #[test] attribute. Test runner with: parallel execution, filtering, output capture, and failure reporting.

6.3 Const and Static Items

Const items: const NAME: Type = expr; defines a compile-time constant. Constants are inlined at each use site. Constants must be evaluable at compile time.

Static items: static NAME: Type = expr; defines a global variable with static lifetime. Static variables have a fixed memory address. Mutable statics require unsafe for access.

Const functions: const fn name(params) -> Type { body } defines a function that can be evaluated at compile time. Const functions are restricted: no heap allocation, no I/O, no loops (in v1.0).

Const generics: fn f[const N: usize](arr: [T; N]) parameterizes on a constant value. Const generics enable: fixed-size arrays, bit widths, and compile-time dimensionality.

Static initialization: statics are initialized before main() runs. Initialization order follows dependency order. Circular static dependencies are compile errors.

Lazy statics: lazy_static!(NAME: Type = expr) initializes on first access. Lazy statics use a Once synchronization primitive. Initialization is thread-safe.

Const evaluation: the compiler evaluates constant expressions at compile time using an interpreter. The interpreter supports: arithmetic, comparison, function calls, and pattern matching.

Const assertions: const_assert!(condition) checks a condition at compile time. Failed const assertions are compile errors. Used for: API contract verification and configuration validation.

Compile-time reflection: const fn type_name[T]() -> &str returns the type name as a string. Compile-time reflection enables: generic formatting and debug printing.

Const evaluation limits: evaluation has a maximum step count (1,000,000 steps) to prevent infinite loops. Exceeding the limit is a compile error with suggestion to simplify.

7.3 Drop and Destructors

Drop trait: trait Drop { fn drop(&mut self); } defines a destructor. The drop method is called when the value goes out of scope. Drop provides deterministic resource cleanup.

Drop order: values are dropped in reverse declaration order. Struct fields are dropped in declaration order. Enum variants drop their contained data. Tuple elements are dropped left-to-right.

Manual drop: std::mem::drop(value) drops a value before the end of its scope. Dropping transfers ownership to the drop function. The variable is moved and no longer usable.

Drop and ownership: only owned values are dropped. Borrowed values are not dropped (the owner handles dropping). Partially moved structs drop only the remaining fields.

Drop flags: the compiler inserts drop flags for conditionally-moved values. The flag is checked at the drop point. If the value has been moved, the drop is skipped.

Drop and unwinding: during panic unwinding, all in-scope values are dropped. Drop implementations should not panic (double panic aborts). Drop during unwinding cleans up resources.

Drop and FFI: values received from foreign functions may require special cleanup. Custom Drop implementations call the foreign library's deallocation function.

No-drop optimization: the compiler detects types whose drop is a no-op (no resources to free). No-drop types skip the drop call, reducing cleanup overhead.

Drop glue: the compiler generates drop glue functions that drop each field of a struct or enum. Drop glue is called automatically; user-defined Drop is optional.

Drop and collections: dropping a Vec drops all its elements, then frees the backing allocation. HashMap drops all key-value pairs, then frees the table. Drop is recursive.

4.3 Array and Slice Expressions

Array literal: [1, 2, 3] creates an array of 3 elements. Repeat: [0; 100] creates an array of 100 zeros. Array type is [T; N] where N is the length.

Array indexing: arr[i] accesses element i. Index is bounds-checked at runtime. Out-of-bounds access panics in debug mode and is undefined behavior in unsafe code.

Array slicing: arr[1..3] creates a slice of elements 1 and 2. arr[..3] takes the first 3. arr[2..] takes from index 2 to end. Slices are borrowed views.

Slice methods: len() returns length. is_empty() checks for zero length. first() and last() return Option. contains() checks for element. sort() sorts in place.

Iterator methods on slices: iter() produces an immutable iterator. iter_mut() produces a mutable iterator. into_iter() consumes the slice (for owned arrays).

Array map: arr.map(|x| x + 1) applies a function to each element, producing a new array. Array map preserves the array length in the type.

Array zip: arr1.zip(arr2) combines two arrays element-wise into an array of tuples. Both arrays must have the same length.

Multi-dimensional arrays: [[T; M]; N] is a 2D array. Indexing: arr[i][j]. Nested array literals: [[1, 2], [3, 4]].

Array to slice coercion: arrays automatically coerce to slices when borrowed. fn f(s: &[i32]) can accept &[1, 2, 3].

Const array initialization: const ARR: [i32; 3] = [1, 2, 3]; creates a compile-time array constant. Const arrays enable lookup tables and configuration data.

5.3 Iterator Protocol

Iterator trait: trait Iterator { type Item; fn next(&mut self) -> Option[Self::Item]; }. Iterators produce a sequence of values. None signals the end.

For loop desugaring: for x in iter { body } is equivalent to: let mut it = iter.into_iter(); while let Some(x) = it.next() { body }.

Iterator adapters: map(f) transforms elements. filter(p) selects elements. enumerate() adds indices. zip(other) combines two iterators. chain(other) concatenates.

Iterator consumers: collect() gathers elements into a collection. sum() adds elements. count() counts elements. any(p) and all(p) test predicates.

Lazy evaluation: iterator adapters are lazy. No computation happens until a consumer is called. Lazy evaluation avoids intermediate allocations.

Double-ended iterators: trait DoubleEndedIterator extends Iterator with next_back(). Enables: rev() (reverse), and consuming from both ends.

Exact-size iterators: trait ExactSizeIterator provides len(). Used by: collect with pre-allocation, and algorithms that need to know the size upfront.

Iterator pipelines: iter.filter(p).map(f).collect() is a pipeline expressed through method chaining. The compiler optimizes iterator pipelines to tight loops.

Custom iterators: implement Iterator for a type to make it iterable. Custom iterators support: file line reading, network message receiving, and tree traversal.

Iterator and ownership: into_iter() consumes the collection (moves elements). iter() borrows (produces &T). iter_mut() mutably borrows (produces &mut T).

9.3 Crate Metadata

Crate manifest: lateralus.toml defines: crate name, version, authors, license, dependencies, build configuration, and features.

Dependency specification: [dependencies] name = '1.0' specifies a dependency with a semver-compatible version. Path dependencies, git dependencies, and registry dependencies are supported.

Feature flags: [features] async = ['dep:tokio'] defines optional features. Features enable conditional compilation and optional dependencies.

Build profiles: [profile.release] optimization_level = 3, debug = false. Profiles configure: optimization, debug info, overflow checks, and LTO.

Workspace support: a workspace contains multiple crates that share: dependencies, build output, and version control. Workspaces enable monorepo development.

Build scripts: build.lat runs before compilation. Build scripts generate: code, configuration, and platform-specific bindings.

Publishing: lateralus-pkg publish uploads the crate to the registry. Published crates are immutable. Yanking hides a version without deleting it.

Version resolution: the resolver selects compatible versions for all dependencies. Conflicts are reported with the dependency chain. lock files ensure reproducible builds.

License compliance: the build tool checks that all dependency licenses are compatible. Incompatible licenses generate warnings or errors depending on configuration.

Crate documentation: lateralus-doc generates HTML documentation from source code.

Documentation is published to the registry alongside the crate.

3.4 User-Defined Types

Struct definition: struct Point { x: f64, y: f64 } defines a struct with two fields. Struct fields are private by default. pub makes fields public.

Tuple struct: struct Color(u8, u8, u8) defines a struct with positional fields. Tuple struct fields are accessed by index: color.0, color.1, color.2.

Unit struct: struct Marker; defines a struct with no fields. Unit structs are used as markers and sentinel values. Unit structs have zero size.

Enum definition: enum Shape { Circle(f64), Rectangle(f64, f64), Triangle(f64, f64, f64) } defines a sum type with three variants carrying data.

Enum variant access: let s = Shape::Circle(5.0); creates a Circle variant. Variants are accessed through the enum name: Shape::Circle, Shape::Rectangle.

Newtype pattern: struct Meters(f64); wraps a primitive type to create a distinct type. Newtypes provide type safety at zero runtime cost.

Recursive types: enum List { Cons(i32, Box[List]), Nil } defines a recursive type using Box for indirection. Direct recursion without Box is a compile error.

Type aliases: type Result[T] = std::result::Result[T, MyError]; creates a shorter name for a common type. Aliases are transparent: they do not create new types.

Struct update syntax: let p2 = Point { x: 1.0, ..p1 }; creates a new Point with x=1.0 and y from p1. The update syntax moves or copies the remaining fields.

Struct and enum derive: #[derive(Debug, Clone, PartialEq)] automatically implements common traits. Derive works for: Debug, Clone, Copy, PartialEq, Eq, Hash, Default, PartialOrd, Ord.

6.4 Unsafe Code

Unsafe blocks: unsafe { code } disables certain safety checks. Unsafe operations include: raw pointer dereference, calling unsafe functions, accessing mutable statics, and implementing unsafe traits.

Raw pointer operations: *ptr reads through a raw pointer. &*ptr converts a raw pointer to a reference. ptr.offset(n) performs pointer arithmetic. All require unsafe.

Unsafe functions: unsafe fn dangerous() { } declares a function that contains unsafe operations. Callers must use unsafe { dangerous() } to call it.

Unsafe traits: unsafe trait GlobalAlloc { } declares a trait whose implementation requires upholding invariants the compiler cannot check. Implementing unsafe traits requires unsafe impl.

FFI: extern 'C' fn handler(arg: i32) -> i32 { } defines a function with C calling convention. extern

blocks declare foreign functions: extern 'C' { fn strlen(s: *const u8) -> usize; }.

Union types: union Value { i: i64, f: f64 } defines an untagged union. Reading union fields requires unsafe because the active field is not tracked.

Inline assembly: unsafe { asm!('nop'); } inserts assembly instructions. Inline assembly is platform-specific and requires unsafe.

Unsafe guidelines: minimize unsafe code. Encapsulate unsafe in safe abstractions. Document invariants that must be maintained. Test unsafe code with Miri (memory safety checker).

Sound abstractions: a safe wrapper around unsafe code is sound if: all safe inputs produce valid behavior, all invariants are maintained, and no undefined behavior is possible through the safe interface.

Unsafe code review: unsafe blocks should be reviewed by multiple developers. Comments explain: why unsafe is needed, what invariants are maintained, and why the code is sound.

7.4 Concurrency and Ownership

Send trait: a type is Send if it can be transferred to another thread. Most types are Send. Types that are not Send: Rc (non-atomic reference count), raw pointers.

Sync trait: a type is Sync if shared references (&T) can be sent to another thread. A type is Sync if &T is Send. Types that are not Sync: Cell, RefCell (non-atomic interior mutability).

Thread spawn: thread::spawn(move || { code }) creates a new thread. The closure must be Send + 'static. Move capture ensures the closure owns its data.

Mutex: Mutex[T] provides mutual exclusion. lock() returns a guard that dereferences to &mut T. The guard releases the lock when dropped. Mutex[T]: Sync when T: Send.

RwLock: RwLock[T] allows multiple readers or one writer. read() returns a shared guard. write() returns an exclusive guard.

Channel: channel() creates a (sender, receiver) pair. send(value) transfers ownership to the receiver. recv() blocks until a value is available.

Arc: Arc[T] enables shared ownership across threads. Arc uses atomic reference counting. Arc[Mutex[T]] is the common pattern for shared mutable state.

Atomic types: AtomicI32, AtomicU64, AtomicBool provide lock-free atomic operations. Methods: load, store, compare_exchange, fetch_add.

Data race freedom: the type system ensures data race freedom. Shared mutable access requires synchronization (Mutex, RwLock, Atomic). Compile-time enforcement prevents data races.

Deadlock prevention: the type system does not prevent deadlocks. Deadlock avoidance is the programmer's responsibility. Tools: lock ordering analysis and deadlock detection at runtime.

8.3 Advanced Patterns

Binding modes: match &x { &y => } vs match &x { y => }. When matching a reference, the pattern can bind by reference (y: &T) or by value (y: T, if T: Copy).

Match ergonomics: when matching a reference, the compiler automatically adds & to patterns. match &Some(42) { Some(x) => } binds x: &i32 without explicit &.

@-bindings: pattern @ name binds the matched value to name while also destructuring. Some(x @ 1..=10) => uses x: i32 where x is between 1 and 10.

Const patterns: patterns can reference constants. const ZERO: i32 = 0; match x { ZERO => 'zero', _ => 'nonzero' }.

Slice patterns: [first, rest @ ..] matches the first element and binds the rest as a slice. [.., last] matches the last element. [a, b, c] matches exactly 3 elements.

Box patterns: match boxed { box value => } destructures a Box and binds the inner value. Box patterns transfer ownership of the inner value.

Multiple match arms: match x { 1 | 2 | 3 => 'small', 4..=10 => 'medium', _ => 'large' }. Or-patterns and ranges enable concise matching.

Nested destructuring: match point { Point { x: 0, y } => 'on y-axis' }. Nested patterns can combine struct, enum, tuple, and literal patterns.

Let-else: let Ok(value) = result else { return Err(e); }; combines pattern matching with early return. The else block must diverge (return, break, continue, or panic).

Pattern in function parameters: fn first((a, _): (i32, i32)) -> i32 { a }. Tuple parameters can be destructured directly in the function signature.

4.4 Struct and Enum Expressions

Struct construction: Point { x: 1.0, y: 2.0 } creates a struct value. All fields must be specified. Field shorthand: Point { x, y } when variable names match field names.

Tuple struct construction: Color(255, 0, 0) creates a tuple struct. Tuple structs use function call syntax.

Enum variant construction: Shape::Circle(5.0) creates an enum variant. Enum variants with named fields: Shape::Rectangle { width: 3.0, height: 4.0 }.

Method call: point.distance(other) calls a method on a value. The compiler automatically borrows or dereferences as needed (auto-ref/deref).

Field access: point.x accesses a struct field. Tuple field access: tuple.0, tuple.1. Field access on references: ref_point.x automatically dereferences.

Struct functional update: Point { x: 1.0, ..other } creates a new struct with x=1.0 and remaining fields

from other. Update syntax moves or copies from the source.

Associated function call: Vec::new() calls an associated function (no self parameter). Associated functions are called through the type name.

Operator expressions: a + b calls the Add trait implementation. a == b calls the PartialEq trait implementation. Operator dispatch is static (no runtime overhead).

Type cast: expr as Type performs a type cast. Casts are available for: numeric types, pointer types, and enum-to-integer. Invalid casts are compile errors.

Sizeof: std::mem::size_of[T]() returns the size of T in bytes. Alignment: std::mem::align_of[T](). Both are evaluated at compile time.

5.4 Loop Expressions

Infinite loop: loop { body }. The only way to exit is break. Loop with value: let x = loop { break 42; }; assigns 42 to x.

While loop: while condition { body }. Condition is evaluated before each iteration. While loops cannot produce values (they return unit).

While let: while let Some(x) = iter.next() { body }. Combines while with pattern matching. The loop continues while the pattern matches.

For loop: for x in 0..10 { body }. The range is consumed by the loop. For loops work with any type implementing IntoIterator.

For loop with pattern: for (i, x) in vec.iter().enumerate() { body }. Destructuring patterns work in for loops.

Break with value: loop { if done { break result; } }. The break value becomes the loop expression's value. All break values must have the same type.

Continue: continue skips the rest of the loop body and starts the next iteration. Continue in for loops advances the iterator.

Labeled loops: 'outer: for x in xs { for y in ys { if condition { break 'outer; } } }. Labels enable breaking or continuing outer loops.

Loop else: for x in iter { if found { break; } } else { not_found_handler(); }. The else block runs when the loop completes without break.

Loop unrolling: the compiler may unroll small loops with known iteration counts. #[unroll(4)] hints the compiler to unroll with factor 4.

9.4 Platform Abstraction

Target specification: each compilation target is described by a target triple:

architecture-vendor-os-environment. Examples: riscv64gc-unknown-linux-gnu, x86_64-unknown-linux-gnu.

Platform-specific code: #[cfg(target_arch = 'riscv64')] enables RISC-V specific code. #[cfg(target_os = 'linux')] enables Linux-specific code.

Conditional dependencies: [target.'cfg(target_os = 'linux')'.dependencies] enables platform-specific dependencies in the manifest.

ABI specification: extern 'C' specifies the C ABI. extern 'lateralus' specifies the Lateralus ABI (default). extern 'system' uses the platform's standard ABI.

Endianness: Lateralus targets little-endian platforms by default. Big-endian support is available through the target specification. Byte order conversion functions are provided.

Pointer size: pointer size depends on the target. cfg(target_pointer_width = '64') detects 64-bit platforms. usize and isize match the pointer size.

Alignment: alignment requirements follow the target ABI. The compiler inserts padding to maintain alignment. #[repr(packed)] removes padding. #[repr(align(N))] increases alignment.

Stack size: the default thread stack size is 2 MB. Custom stack sizes are specified when spawning threads. Stack overflow is detected at runtime.

Floating-point: targets without FPU support use software floating-point emulation. The compiler selects soft-float or hard-float based on the target specification.

Embedded targets: targets without an OS use the no_std attribute. no_std programs do not link the standard library. A minimal core library provides: types, traits, and basic operations.

2.3 Lexical Rules

Whitespace characters: space (U+0020), horizontal tab (U+0009), newline (U+000A), carriage return (U+000D). Whitespace separates tokens but is otherwise insignificant.

Line comments start with // and extend to the end of the line. Block comments start with /* and end with */. Block comments nest.

Token disambiguation: > followed by > is two greater-than tokens (not right-shift) in generic contexts. The parser disambiguates based on context.

Reserved words: the following identifiers are reserved for future use: yield, abstract, final, override, macro, do, try, catch, finally, throw. Using reserved words as identifiers is an error.

Raw identifiers: r#keyword uses a keyword as an identifier. Raw identifiers are useful for: FFI interop with languages that use different keywords, and migration compatibility.

Character literals: 'a' is a character literal. Escape sequences: '\n' (newline), '\t' (tab), '\\' (backslash), '\'' (quote), '\0' (null), '\u{1F600}' (Unicode). Characters are Unicode scalar values.

Byte literals: b'a' is a byte literal (u8). Byte string literals: b'hello' has type &[u8; 5]. Byte literals support the same escape sequences as character literals (except Unicode).

Lifetime tokens: 'a is a lifetime. Lifetimes start with a single quote followed by an identifier. 'static is the static lifetime. '_ is the anonymous lifetime.

Shebang: #! at the start of a file is ignored (for Unix script execution). The shebang line is stripped before lexing.

Operator tokens are maximal: >>= is parsed as >>= (right-shift-assign), not > >= (greater, greater-equal). The lexer always chooses the longest matching token.

3.5 Never Type

The never type (!) represents computations that never complete. Functions that diverge (always panic or loop forever) return !. The never type coerces to any type.

Examples: fn exit(code: i32) -> ! { process::exit(code) } returns never. loop {} has type !. panic!('msg') has type !.

Never in match: match x { 0 => 'zero', _ => panic!('nonzero') } type-checks because panic returns ! which coerces to &str.

Never in Result: Result[T, !] represents a result that never fails. Infallible operations return this type. unwrap() on Result[T, !] is guaranteed safe.

Empty enums: enum Void {} has no variants and no values. Void is isomorphic to !. Matching on Void requires no arms.

Never type coercion: ! coerces to any type T. This enables: diverging branches in if-else, panic in match arms, and return in closures.

Never and type inference: when the compiler infers !, it means the code is unreachable. Unreachable code after ! expressions is dead code.

Never in generics: a generic function returning T can be called with ! for T. The function call never returns, which is consistent with T = !.

Never and closures: a closure that always panics has return type !. The closure can be used where any return type is expected.

Never stability: the never type (!) is stabilized in Lateralus 1.0. It can be used in: function signatures, type annotations, and generic bounds.

6.5 Impl Blocks Detailed

Inherent impl: impl TypeName { methods } adds methods to a type. Inherent methods are called with method syntax: value.method(). Each type can have multiple impl blocks.

Self type: within an impl block, Self refers to the implementing type. fn new() -> Self creates a new instance. fn take(self) consumes the instance.

Method receivers: &self (shared borrow), &mut self (exclusive borrow), self (consume), Box[Self] (consume boxed). The receiver determines how the method accesses the instance.

Static methods: methods without a self parameter are static. Called through the type: Type::method(). Constructors are typically static methods named new.

Generic impl blocks: impl[T] Vec[T] { } implements methods for all Vec types. impl[T: Display] Vec[T] { fn print(&self) { } } implements only for Vec of printable types.

Impl for external types: traits defined in the current crate can be implemented for external types. External traits cannot be implemented for external types (orphan rule).

Negative impls: impl !Send for MyType {} explicitly marks a type as not Send. Negative impls override automatic trait implementations.

Impl overlap: two impl blocks cannot provide the same method for the same type. Overlapping impls are detected at compile time and reported as errors.

Auto traits: Send, Sync, Unpin are auto-implemented based on the type's fields. A struct is Send if all fields are Send. Auto traits propagate transitively.

Deref impl: impl Deref for Smart { type Target = T; fn deref(&self) -> &T { } } enables smart pointer dereferencing. Deref coercion automatically borrows through the pointer.

7.5 Memory Model

Lateralus uses a sequentially consistent memory model for atomic operations by default. Relaxed orderings are available through explicit ordering parameters.

Memory ordering options: Relaxed (no ordering guarantee), Acquire (reads after this see writes before the corresponding Release), Release (writes before this are visible after corresponding Acquire), SeqCst (total ordering).

Happens-before relation: if operation A happens-before operation B, then A's effects are visible to B. Thread creation, join, mutex operations, and atomic operations establish happens-before.

Data race: simultaneous access to the same memory location where at least one access is a write and no ordering relation exists. Data races are undefined behavior.

The compiler enforces absence of data races through the type system. Shared mutation requires synchronization (Mutex, RwLock, Atomic). The Send and Sync traits encode thread safety.

Volatile access: volatile reads and writes are used for memory-mapped I/O. Volatile operations are not optimized away. Volatile access requires unsafe.

Memory allocation: objects are allocated with a global allocator. The default allocator wraps the

system malloc. Custom allocators implement the GlobalAlloc trait.

Stack allocation: local variables and small temporaries are stack-allocated. The compiler determines stack size at compile time. Variable-length arrays are heap-allocated.

Zero-cost abstraction: ownership, borrowing, and lifetime checks are purely compile-time. No runtime overhead for safety checks (except bounds checks, which can be disabled with unsafe).

Undefined behavior: Lateralus defines undefined behavior for: data races, invalid raw pointer access, incorrect unsafe trait implementations, and violation of aliasing rules. UB enables compiler optimizations.

8.4 Pattern Matching Optimization

Decision tree compilation: the compiler converts match expressions to decision trees. Each internal node tests a value. Each leaf executes an arm's body.

Constructor testing: enum variants are tested by comparing the discriminant. Integer patterns use equality comparison. Range patterns use bounds comparison.

Shared subtrees: when multiple arms share the same prefix, the decision tree shares the prefix nodes. Sharing reduces generated code size.

Default arms: the wildcard arm becomes the default branch at each decision node. Default branches are taken when no other constructor matches.

Guard evaluation: guards are evaluated after the pattern matches. If the guard fails, matching continues with the next arm. Guards prevent unconditional commitment to an arm.

Match on references: matching &T automatically follows the reference. The matched value is the referent, not the pointer. This enables ergonomic matching on borrowed data.

Match compilation performance: the compiler generates at most O(n * m) tests where n is the number of arms and m is the pattern depth. Typical matches generate fewer tests due to sharing.

Jump table optimization: match on contiguous integer ranges generates a jump table. Jump table lookup is O(1). The compiler selects jump tables for ranges of 4 or more consecutive values.

Binary search optimization: match on non-contiguous integer values generates a binary search tree. Binary search is O(log n). Used when jump tables would be too sparse.

Match arm ordering: the compiler reorders arms for: fewer tests, better branch prediction, and smaller code. User-visible semantics (first match wins) are preserved.

4.5 String Expressions

String literals: 'hello' creates a &str (string slice) with static lifetime. String literals are stored in the executable's read-only data section.

String interpolation: f'Hello, {name}!' creates a formatted string. Expressions in braces are evaluated and converted to strings using the Display trait.

Raw strings: r'no \n escapes' creates a string without escape processing. Raw strings are useful for: regular expressions, file paths, and SQL queries.

Multi-line strings: '''content''' creates a multi-line string. Leading whitespace up to the closing delimiter is stripped. Multi-line strings preserve newlines.

String concatenation: 'hello' + ' ' + 'world' concatenates strings. The + operator requires String on the left and &str on the right. Use format!() for complex formatting.

String methods: len() (byte length), chars() (character iterator), contains(pat), starts_with(pat), ends_with(pat), replace(from, to), split(pat), trim(), to_uppercase(), to_lowercase().

String slicing: &s[0..5] creates a byte-offset slice. Slicing at non-character boundaries panics. Use char_indices() for safe character-boundary slicing.

String conversion: to_string() converts any Display type to String. parse() converts a string to any FromStr type. Both return results for fallible conversions.

String formatting: format!('x={}, y={}', x, y) creates a formatted string. Format specifiers: {:?} (Debug), {:#?} (pretty Debug), {:x} (hex), {:b} (binary), {:e} (scientific).

String encoding: strings are UTF-8 encoded. The encoding is validated on construction. as_bytes() provides access to the raw bytes. from_utf8() validates and converts bytes to string.

10.1 Appendix: Grammar Summary

Module := Item*. Item := FnDef | StructDef | EnumDef | TraitDef | ImplBlock | UseDef | ConstDef | StaticDef | ModDef.

FnDef := 'fn' IDENT GenericParams? '(' Params ')' ('->' Type)? Block. Params := (Pattern ':' Type ',')*.

StructDef := 'struct' IDENT GenericParams? ('{' Fields '}' | '(' Types ')' ';' | ';'). Fields := (IDENT ':' Type ',')*.

EnumDef := 'enum' IDENT GenericParams? '{' Variants '}'. Variants := (IDENT ('(' Types ')' | '{' Fields '}')? ',')*.

TraitDef := 'trait' IDENT GenericParams? (':' TraitBounds)? '{' TraitItem* '}'. TraitItem := FnSig ';' | FnDef.

ImplBlock := 'impl' GenericParams? (TraitPath 'for')? Type '{' ImplItem* '}'. ImplItem := FnDef | ConstDef | TypeAlias.

Expr := LitExpr | PathExpr | BlockExpr | IfExpr | MatchExpr | LoopExpr | PipeExpr | BinExpr | UnExpr | CallExpr | FieldExpr | IndexExpr | CastExpr.

Stmt := LetStmt | ExprStmt | ReturnStmt. LetStmt := 'let' 'mut'? Pattern (':' Type)? '=' Expr ';'.

Pattern := LitPat | IdentPat | WildPat | TuplePat | StructPat | EnumPat | RefPat | RangePat | OrPat | GuardPat.

Type := PrimType | TupleType | ArrayType | SliceType | RefType | FnType | PathType | NeverType. PrimType := 'i8' | 'i16' | 'i32' | 'i64' | 'u8' | 'u16' | 'u32' | 'u64' | 'f32' | 'f64' | 'bool' | 'char' | 'unit'.

10.2 Appendix: Keyword Table

Control flow keywords: if, else, match, while, for, loop, break, continue, return. These keywords control execution order and branching.

Declaration keywords: fn, let, mut, struct, enum, trait, impl, mod, use, const, static, type, where. These keywords introduce new definitions.

Ownership keywords: ref, move, self, Self. ref creates a reference pattern. move forces closure capture by value. self is the method receiver. Self is the implementing type.

Visibility keywords: pub, pub(crate), pub(super). pub makes items public. pub(crate) restricts to the crate. pub(super) restricts to the parent module.

Safety keywords: unsafe, extern. unsafe enables unchecked operations. extern specifies foreign function interfaces and calling conventions.

Async keywords: async, await. async marks asynchronous functions. await suspends until a future completes. Async is planned for version 1.1.

Boolean keywords: true, false. Boolean literals of type bool.

Pipeline keywords: pipe. The pipe keyword introduces pipeline definitions. The |> operator creates pipeline expressions.

Type keywords: as performs type casting. in is used in for-in loops. The : operator separates names from types.

Placeholder keywords: _ is the wildcard pattern and placeholder. It matches any value and discards it. _ in expressions is an inference placeholder.

10.3 Appendix: Operator Precedence Table

Level 1 (highest): Unary operators: - (negation), ! (not), & (borrow), &mut (mutable borrow), * (dereference). Unary operators are prefix and right-associative.

Level 2: Method call and field access: expr.method(), expr.field. Left-associative. Auto-deref applies through reference and smart pointer chains.

Level 3: Function call and indexing: f(args), arr[index]. Left-associative. Calling through function pointers uses the same syntax.

Level 4: Type cast: expr as Type. Left-associative. Casts between numeric types, pointer types, and

enum-to-integer.

Level 5: Multiplicative: * (multiply), / (divide), % (remainder). Left-associative. Operates on numeric types.

Level 6: Additive: + (add), - (subtract). Left-associative. String concatenation uses + on String and &str.

Level 7: Shift: << (left shift), >> (right shift). Left-associative. Shift amount must be non-negative and less than the bit width.

Level 8: Bitwise AND: &. Left-associative. Also used for borrow in expression context (disambiguated by parser).

Level 9: Bitwise XOR: ^. Left-associative.

Level 10: Bitwise OR: |. Left-associative. Not to be confused with the pipeline operator |>.

Level 11: Comparison: ==, !=, <, >, <=, >=. Non-associative (cannot chain: a < b < c is an error). Returns bool.

Level 12: Logical AND: &&. Left-associative. Short-circuit: right operand is not evaluated if left is false.

10.4 Appendix: Type Conversion Rules

Implicit widening: i8 -> i16 -> i32 -> i64. u8 -> u16 -> u32 -> u64. f32 -> f64. Widening conversions never lose precision.

Explicit narrowing: i64 as i32 truncates. u64 as u32 truncates. f64 as f32 rounds. Narrowing may lose precision or change sign.

Integer to float: i32 as f64 is exact (f64 has 53-bit mantissa). i64 as f64 may lose precision (values above 2^53). i32 as f32 may lose precision (values above 2^24).

Float to integer: f64 as i64 truncates toward zero. NaN converts to 0. Infinity saturates to the maximum/minimum value. Out-of-range values saturate.

Bool conversions: true as i32 == 1. false as i32 == 0. Integer to bool is not allowed (use != 0 explicitly).

Pointer conversions: *const T as *const U reinterprets the pointer type. *const T as usize converts to an integer. usize as *const T converts from an integer. All require unsafe.

Deref coercion: &String coerces to &str. &Vec[T] coerces to &[T]. &Box[T] coerces to &T. Deref coercion follows the Deref trait chain.

Unsized coercion: [T; N] coerces to [T]. T coerces to dyn Trait (creating a trait object). Unsized coercions happen at borrow sites.

From/Into traits: From[T] for U enables U::from(t). Into[U] for T is automatically provided. Use Into in generic bounds for flexibility.

TryFrom/TryInto: TryFrom[T] for U returns Result. Used for fallible conversions. TryFrom[i64] for i32 returns Err for values outside i32 range.

AsRef/AsMut: AsRef[T] for U provides cheap reference conversions. String: AsRef[str], Vec[T]: AsRef[[T]]. Used in generic function parameters.

Display and ToString: Display provides user-facing string formatting. Any Display type automatically gets ToString. to_string() calls Display::fmt.

3.6 Trait Object Types

Trait object type: dyn Trait is a dynamically-sized type representing any value implementing Trait. Trait objects are always used behind a pointer: &dyn Trait, Box[dyn Trait].

Fat pointer: a trait object reference is 16 bytes (data pointer + vtable pointer). The vtable contains: destructor, size, alignment, and method pointers.

Object safety requirements: all methods must take self by reference (&self or &mut self). Methods cannot be generic. Methods cannot return Self.

Multiple traits: dyn Trait1 + Trait2 requires both traits. The vtable contains methods from both traits. At most one non-auto trait is allowed.

Lifetime bounds: dyn Trait + 'a specifies that the underlying value lives at least as long as 'a. The default bound is 'static for Box[dyn Trait].

Upcasting: &dyn SubTrait can be converted to &dyn SuperTrait. The vtable pointer is adjusted to the supertrait's vtable.

Trait object creation: &value as &dyn Trait creates a trait object from a concrete value. The compiler generates the vtable at compile time.

Trait object dispatch: method calls on trait objects use indirect dispatch (vtable lookup). The overhead is one pointer indirection per method call.

Any trait: dyn Any enables runtime type identification. downcast_ref() attempts to recover the concrete type. Returns None if the types do not match.

Trait object patterns: match statements on trait objects require downcast. Direct pattern matching on trait objects is not supported (the concrete type is unknown).

Performance of trait objects: vtable dispatch adds approximately 5 nanoseconds per call compared to static dispatch. The overhead is due to: indirect branch and potential cache miss.

Trait objects vs generics: generics provide static dispatch (monomorphization, zero overhead). Trait objects provide dynamic dispatch (one compiled version, slight overhead). Choose based on

flexibility vs performance needs.

6.6 Attribute Reference

Derive attributes: #[derive(Debug)] generates Debug implementation. Derivable traits: Debug, Clone, Copy, PartialEq, Eq, Hash, Default, PartialOrd, Ord.

Cfg attributes: #[cfg(target_os = 'linux')] includes item on Linux. #[cfg(feature = 'async')] includes when feature is enabled. cfg_attr applies attributes conditionally.

Test attributes: #[test] marks test functions. #[bench] marks benchmark functions. #[ignore] skips tests. #[should_panic(expected = 'msg')] expects a panic.

Documentation attributes: #[doc = 'text'] is equivalent to /// text. #[doc(hidden)] hides from documentation. #[doc(alias = 'name')] adds search aliases.

Representation: #[repr(C)] C-compatible layout. #[repr(transparent)] single-field struct has same layout as the field. #[repr(u8)] sets enum discriminant type.

Optimization: #[inline] suggests inlining. #[cold] marks rarely called functions. #[target_feature(enable = 'avx2')] enables CPU features.

Diagnostic: #[allow(warning)] suppresses warning. #[warn(warning)] enables warning. #[deny(warning)] makes warning an error. #[forbid(warning)] same as deny, cannot be overridden.

Linking: #[link(name = 'lib')] specifies library to link. #[link_name = 'name'] specifies symbol name. #[no_mangle] preserves function name for FFI.

Stability: #[stable(since = '1.0')] marks stable API. #[unstable(feature = 'name')] marks experimental API. #[deprecated] marks removed API.

Macro: #[macro_export] exports a macro. #[macro_use] imports macros from a crate. #[proc_macro] marks a procedural macro function.

Safety: #[must_use] warns if return value is unused. #[non_exhaustive] prevents external code from constructing or exhaustively matching. #[may_dangle] for advanced drop checking.

Alignment: #[align(N)] sets minimum alignment. #[packed] removes padding. #[repr(align(4096))] aligns to page boundary for memory-mapped structures.

10.5 Appendix: Reserved Symbols

Reserved operator symbols for future use: ~, `, ?, @, $, #. These symbols are used in specific contexts (? for error propagation, # for attributes) but are reserved as standalone operators.

The @ symbol is used in pattern bindings: name @ pattern. Future use may extend @ for: decorators, annotation syntax, or compile-time evaluation.

The $ symbol is reserved for macro metavariables: $name in macro definitions. Future use may

extend $ for: template strings or shell integration.

The # symbol prefixes attributes: #[attr] and #![attr]. The # symbol cannot be used as an operator or in identifiers.

The ? operator performs error propagation on Result and Option types. expr? unwraps Ok/Some or returns Err/None. The ? operator is not available as a general postfix operator.

The .. operator creates ranges: 1..10 (exclusive), 1..=10 (inclusive). The ... syntax (three dots) is reserved and produces an error suggesting .. or ..= instead.

The => symbol separates patterns from bodies in match arms. => is not available as a general operator.

The -> symbol separates parameter lists from return types in function signatures. -> is not available as a general operator.

The :: symbol is the path separator for module paths and associated items. Type::method, module::item. :: is not available as a general operator.

The |> symbol is the pipeline operator. |> is unique to Lateralus and distinguishes it from other systems languages. Pipeline syntax is the language's defining feature.

The <> symbols in generic contexts: Vec[T] uses brackets (not angle brackets) for generic parameters. This avoids the parsing ambiguity of < and > as both comparison and generic delimiters.

Future syntax extensions: the language reserves the right to add new operators and syntax in future versions. User code using reserved symbols will receive deprecation warnings before breaking changes.

References

[1] Pierce, B. Types and Programming Languages. MIT Press, 2002.
[2] Harper, R. Practical Foundations for Programming Languages. Cambridge, 2016.
[3] Milner, R. et al. The Definition of Standard ML. MIT Press, 1997.
[4] The Rust Reference. The Rust Project Developers, 2024.
[5] ISO/IEC 9899:2018. Programming Languages - C. ISO, 2018.