From Lexer to Language: Building the Lateralus Compiler

bad-antics | January 2024 | Compilers

Abstract

This paper traces the complete compilation path from Lateralus source text to executable machine code, covering lexical analysis, parsing, name resolution, type checking, borrow checking, intermediate representation, optimization, code generation, and linking. Each compiler phase is examined with attention to the unique challenges of a pipeline-native language with ownership semantics.

1 Introduction

This paper traces the complete path from source text to executable code in the Lateralus compiler. We examine each compiler phase in detail: lexical analysis, parsing, semantic analysis, type checking, intermediate representation, optimization, and code generation. The goal is to provide a comprehensive understanding of how a modern systems language compiler transforms human-readable source into efficient machine code.

Compiler construction is a deep and well-studied field, but the specific design decisions for a pipeline-native language with ownership semantics present unique challenges. This paper documents those challenges and the solutions implemented in the Lateralus compiler.

2 Lexical Analysis

The lexer converts a stream of Unicode characters into a stream of tokens. Each token has a type (keyword, identifier, literal, operator, punctuation), a value (the actual text), and a source location (file, line, column).

Keyword recognition uses a perfect hash function computed at compile time. The hash function maps keyword strings to token types in a single lookup without collisions. Non-keyword identifiers fall through to the identifier token type.

String literal lexing handles escape sequences (backslash-n, backslash-t, Unicode escapes), raw strings (no escape processing), and multi-line strings (preserving indentation). String interpolation is desugared into concatenation during lexing.

Numeric literal lexing supports decimal, hexadecimal (0x), octal (0o), and binary (0b) bases. Underscores in numeric literals are ignored, allowing 1_000_000 for readability. Floating-point literals use the standard notation with optional exponent.

Comment handling strips single-line comments (//) and block comments (/* */). Doc comments (///) are preserved as tokens for the documentation generator. Nested block comments are supported.

```
  // Lexer output for a pipeline expression
```

```
// Source: data |> filter(|x| x > 0) |> map(|x| x * 2)
// Tokens:
//   IDENT("data")
//   PIPE            // |>
//   IDENT("filter")
//   LPAREN
//   PIPE_ARG        // |
//   IDENT("x")
//   PIPE_ARG        // |
//   IDENT("x")
//   GT
//   INT(0)
//   RPAREN
//   PIPE            // |>
//   IDENT("map")
//   ...
```

3 Parsing

The parser transforms the token stream into an abstract syntax tree (AST). Lateralus uses a recursive descent parser with Pratt parsing for expressions. The parser is hand-written for precise error messages and error recovery.

Pratt parsing (top-down operator precedence) handles binary operators, unary operators, and the pipeline operator with correct precedence and associativity. The pipeline operator |> has lower precedence than arithmetic operators but higher precedence than assignment.

Error recovery uses synchronization tokens (semicolons, closing braces) to resume parsing after an error. The parser reports the error and skips tokens until a synchronization point, then continues parsing the rest of the file.

Expression parsing handles literals, variables, function calls, method calls, field access, index access, closures, if-else, match, and block expressions. Each expression type has a dedicated parsing function.

Statement parsing handles let bindings, assignments, expression statements, return, break, continue, and loop statements. Statements are terminated by semicolons, with implicit semicolons at the end of block expressions.

Item parsing handles function definitions, struct definitions, enum definitions, trait definitions, impl blocks, and module declarations. Items form the top-level structure of the program.

4 Name Resolution

Name resolution maps each identifier in the AST to its definition. The resolver maintains a scope stack, pushing a new scope for each block, function, module, and impl block. Lookups search from the innermost scope outward.

Module resolution follows the module hierarchy. A qualified name like std::collections::HashMap is resolved by traversing the module tree. Use declarations import names into the current scope.

Trait method resolution determines which trait implementation to call for a method invocation. The resolver considers the receiver type, the method name, and the traits in scope to select the unique matching implementation.

Lifetime resolution assigns lifetime parameters to references. The resolver creates fresh lifetime variables for elided lifetimes and records the relationships between lifetime parameters.

5 Type Checking

Type checking verifies that every expression has a consistent type and that all operations are applied to compatible types. The type checker uses bidirectional type inference: types flow both from expressions to their contexts and from contexts to expressions.

Constraint generation creates type constraints from each expression. A function call generates constraints equating the argument types with the parameter types. An arithmetic operation generates constraints requiring numeric types.

Constraint solving uses unification to find the most general type assignment satisfying all constraints. The solver handles type variables, generic instantiation, and trait bounds. Unsolvable constraints produce type error messages.

Trait checking verifies that types implement the required traits. Trait bounds on generic parameters are checked against the available implementations. The trait checker handles associated types and supertraits.

Ownership checking verifies the ownership rules: each value has one owner, borrows do not outlive owners, and mutable borrows are exclusive. The borrow checker runs after type checking and uses the type information.

```
// Type checking a pipeline
// data: Vec<i32>
// data |> filter(|x| x > 0) |> map(|x| x * 2) |> collect::<Vec<i32>>()
//
// Constraints generated:
//   typeof(data) = Vec<i32>
//   filter input:  Iterator<Item = i32>
//   filter pred:   Fn(&i32) -> bool
//   map input:     Iterator<Item = i32>
//   map func:      Fn(i32) -> i32
//   collect output: Vec<i32>
```

6 Intermediate Representation

The compiler lowers the typed AST into a control-flow graph (CFG) based intermediate

representation. Each function is represented as a sequence of basic blocks containing typed instructions. The IR uses static single assignment (SSA) form.

SSA form ensures that each variable is defined exactly once. Phi nodes at control flow merge points select the correct definition based on which predecessor block was executed. SSA simplifies many optimization passes.

IR instructions include arithmetic, comparison, load, store, call, branch, return, and phi. Each instruction has typed operands and a typed result. The IR type system is simpler than the source language type system.

Drop insertion adds explicit drop calls for owned values at the end of their scope. The ownership analysis determines the drop point for each value. Drop calls are represented as IR function calls to the type's drop method.

7 Optimization

The optimization pipeline applies a sequence of transformation passes to the IR. Each pass reads the IR, transforms it, and produces a new IR. The pass order is chosen to maximize the effectiveness of each pass.

Dead code elimination removes instructions whose results are never used. The analysis propagates from roots (function returns, side-effecting calls) to mark live instructions. Unmarked instructions are removed.

Constant propagation replaces variable uses with their known constant values. The analysis tracks constant values through the CFG, handling phi nodes by computing the meet of incoming values.

Inlining replaces function calls with the body of the called function. Inlining decisions use a cost model that considers function size, call frequency, and the optimization opportunities created by inlining.

Pipeline fusion merges adjacent pipeline stages into a single loop. The fusion pass recognizes pipeline patterns in the IR and transforms them into fused loop bodies. Fusion eliminates intermediate iterator overhead.

8 Code Generation

Code generation translates the optimized IR into machine code for the target architecture. The code generator performs instruction selection, register allocation, and instruction scheduling.

Instruction selection maps IR instructions to machine instructions. The selector uses tree pattern matching to find the most efficient instruction sequence for each IR pattern. Complex IR patterns may map to single machine instructions.

Register allocation assigns physical registers to IR virtual registers. The allocator uses linear scan for

fast compilation and graph coloring for optimized compilation. Spilled values are stored on the stack.

Instruction scheduling reorders machine instructions to maximize pipeline utilization and minimize stalls. The scheduler considers instruction latencies, dependencies, and resource constraints.

9 Conclusion

This paper traced the complete path from Lateralus source code to machine code, covering every major compiler phase. The pipeline-native design and ownership semantics introduce unique compiler engineering challenges that are addressed through specialized analysis passes.

2.1 Lexer Performance

The lexer processes approximately 500 MB/s of source text on modern hardware. Performance is achieved through branch-free character classification using lookup tables and SIMD-accelerated string comparison for keywords.

Token allocation uses an arena allocator that bulk-allocates token structures. The arena is freed after parsing completes, avoiding per-token deallocation overhead. The arena approach reduces allocation time by 85% compared to individual allocation.

Incremental lexing re-lexes only the modified portion of a file. The incremental lexer uses a checkpoint system that records the lexer state at line boundaries. Editing a line re-lexes from the nearest checkpoint.

Unicode handling uses the UTF-8 encoding throughout. Identifier characters are validated against the Unicode XID_Start and XID_Continue categories. The validation uses a binary search on the Unicode category table.

Token interning stores each unique identifier string once in a string interner. Token values reference the interned string by index, reducing memory usage for files with repeated identifiers. Interning also enables O(1) identifier comparison.

Whitespace handling is context-sensitive: indentation is significant in some contexts (multi-line strings) but ignored in others (expression continuation). The lexer tracks the indentation context stack.

Lookahead management: the lexer uses at most 2 characters of lookahead, which is sufficient to distinguish all token types. The two-character lookahead enables recognizing operators like |> (pipe), >> (right shift), and >= (greater-equal).

Error token generation: when the lexer encounters an invalid character, it produces an error token and advances past the invalid character. Error tokens are reported to the user and skipped by the parser.

Compile-time lexer generation: the lexer's keyword table and character classification tables are generated at compile time using const functions. The generated tables are embedded in the compiler

binary as static data.

Parallel lexing splits large files into chunks at line boundaries and lexes each chunk independently. The chunks are merged, with fixup for tokens that span chunk boundaries. Parallel lexing provides 3.5x speedup on 4 cores.

3.1 AST Design

The AST uses interned identifiers and compressed source locations. Each AST node stores a 32-bit source span (packed file ID, start line, and length). The compressed representation reduces AST memory usage by 40%.

AST nodes use tagged unions (Rust-style enums) for expression, statement, and item variants. Each variant stores its specific fields inline. The enum representation avoids heap allocation for small nodes.

Expression nodes include: Literal(value), Ident(name), Binary(op, left, right), Unary(op, expr), Call(func, args), Pipeline(stages), Match(scrutinee, arms), If(cond, then, else), Block(stmts).

Statement nodes include: Let(pattern, type, init), Assign(lhs, rhs), Expr(expr), Return(expr), Break(expr), Continue, Loop(body), While(cond, body), For(pattern, iter, body).

Item nodes include: Fn(name, params, ret_type, body), Struct(name, fields), Enum(name, variants), Trait(name, methods), Impl(type, trait, methods), Mod(name, items), Use(path).

AST pretty-printing reproduces the source code from the AST. The pretty-printer respects indentation, line width limits, and comment placement. Pretty-printing is used for code formatting and refactoring output.

AST visitors implement the visitor pattern for tree traversal. The visitor trait provides default methods that recurse into child nodes. Overriding specific methods customizes the traversal.

AST transformation uses a fold trait that rebuilds the tree with modifications. Each fold method returns a new node that replaces the original. The fold pattern is used for desugaring and macro expansion.

AST serialization saves the tree to a binary format for incremental compilation. The serialized format uses variable-length integers and string interning for compact representation.

AST hashing computes content hashes for incremental compilation. If a module's AST hash matches the cached hash, the module can skip re-compilation. Hashing is faster than full comparison.

5.1 Borrow Checking

The borrow checker operates on the MIR (Mid-level IR), which represents the program as a control flow graph with explicit moves, borrows, and drops. The MIR provides the precise control flow information needed for borrow analysis.

Liveness analysis determines which variables are live (potentially used in the future) at each program point. A variable is live if there exists a path from the current point to a use of the variable without an intervening definition.

Borrow tracking records each active borrow: the borrowed place (variable, field, or index), the borrow kind (shared or mutable), and the borrow's lifetime. Active borrows are checked against new borrows and mutations.

Conflict detection checks each new borrow and mutation against active borrows. A mutable borrow conflicts with any other borrow of the same place. A mutation conflicts with any borrow of the same place.

Two-phase borrows handle the common pattern of calling a method that takes &mut self while borrowing the return value. The first phase creates a reserved borrow that does not conflict with shared borrows. The second phase activates the mutable borrow.

Non-lexical lifetimes (NLL) end borrows at their last use rather than at the end of the lexical scope. NLL analysis uses the control flow graph to determine the minimal lifetime for each borrow, accepting more programs.

Polonius is the next-generation borrow checker that uses a Datalog-based formulation. Polonius handles more complex borrowing patterns by computing the precise set of loans that are active at each program point.

Move checking verifies that moved values are not used after the move. The checker tracks which places contain valid values and reports uses of moved values. Partial moves of struct fields are tracked independently.

Error reporting for borrow checker violations includes the conflicting borrows, their locations, and suggestions for fixing the conflict. Common fixes include restructuring code, using cloning, or using interior mutability.

Borrow checker performance is O(n * m) where n is the number of program points and m is the number of borrows. For typical functions, the borrow checker completes in microseconds.

6.1 IR Lowering

AST-to-IR lowering translates high-level constructs into primitive operations. Match expressions are lowered into decision trees of conditional branches. Loops are lowered into basic blocks with backedges.

Closure lowering captures the closure's free variables in a struct. The closure struct contains the captured values (moved) or references (borrowed). The closure call is lowered into a function call with the closure struct as an extra parameter.

Generic instantiation creates specialized copies of generic functions for each type argument. Monomorphization produces type-specific code that the optimizer can specialize. Unused generic

instantiations are removed.

Trait object lowering creates vtables (virtual method tables) for trait objects. Each trait implementation generates a vtable containing function pointers for the trait's methods. Trait object calls use indirect calls through the vtable.

Pipeline lowering translates pipeline expressions into iterator-based loops. Each pipeline stage is lowered into a closure that processes one element. The pipeline loop calls each closure in sequence.

Destructor insertion adds drop calls at the end of each variable's scope. The drop elaboration pass handles conditional drops (values that may or may not be moved) using drop flags.

Panic lowering converts panic expressions into calls to the panic handler. The panic handler prints the message and unwinds the stack, calling destructors for each stack frame.

Async lowering transforms async functions into state machines. Each await point becomes a state transition. The state machine implements the Future trait, allowing the runtime to drive execution.

Debug info generation creates source location mappings from IR instructions to source code positions. The mappings enable source-level debugging with breakpoints, stepping, and variable inspection.

IR validation checks the lowered IR for consistency: every use is dominated by its definition, every branch targets a valid block, and every instruction has correctly typed operands.

7.1 Advanced Optimizations

Loop invariant code motion moves computations that produce the same result on every iteration out of the loop. The analysis identifies instructions whose operands are defined outside the loop or are themselves loop-invariant.

Strength reduction replaces expensive operations with cheaper ones. Multiplication by a constant is replaced with shifts and additions. Division by a power of two is replaced with a right shift.

Escape analysis determines whether an allocated value can be observed outside its creating function. Non-escaping allocations can be placed on the stack instead of the heap, eliminating allocation overhead.

Devirtualization replaces indirect (virtual) calls with direct calls when the concrete type is known. The analysis uses type information from the class hierarchy and value flow analysis.

Tail call optimization converts recursive calls in tail position into jumps. Tail call optimization prevents stack overflow in recursive functions and enables recursive algorithms to use constant stack space.

Vectorization converts scalar operations into SIMD operations. The vectorizer identifies loops with independent iterations that can be processed in parallel using vector instructions.

Partial evaluation evaluates parts of the program at compile time. Partial evaluation specializes generic code for known arguments, removing abstraction overhead.

Interprocedural optimization analyzes multiple functions together. Interprocedural constant propagation, alias analysis, and escape analysis provide more precise information than intraprocedural analysis.

Profile-guided optimization uses runtime profiling data to direct optimization decisions. Hot paths are optimized aggressively (inlining, unrolling), while cold paths are optimized for size.

Link-time optimization applies optimizations across compilation unit boundaries. Cross-module inlining and dead code elimination improve performance and reduce binary size.

4.1 Name Resolution Details

Scope resolution uses a stack of hash maps. Each scope maps names to definitions. Function parameters, local variables, and loop variables are inserted into the innermost scope. Block exit pops the scope.

Forward reference resolution allows functions to call other functions defined later in the same module. The resolver performs two passes: the first collects all item definitions, and the second resolves references.

Import resolution follows use declarations to their targets. Glob imports (use mod::*) import all public names from the target module. Selective imports (use mod::{A, B}) import specific names.

Shadowing allows inner scopes to redefine names from outer scopes. The inner definition takes precedence within its scope. The outer definition is restored when the inner scope exits. Shadowing is allowed but produces a warning for function parameters.

Generic name resolution resolves type parameters and trait bounds. Type parameters are introduced by function signatures and impl blocks. Trait bounds constrain the set of types that can be used for each parameter.

Macro name resolution handles hygienic macros that do not capture names from the call site. Macro-generated names are in a separate namespace from user-written names. Hygiene prevents accidental name collisions.

Associated name resolution resolves names within impl blocks and trait definitions. Method names, associated type names, and associated constant names are resolved relative to the implementing type and trait.

Error recovery in name resolution reports undefined names and suggests similar names from the current scope. The suggestion algorithm uses edit distance to find the closest matching name.

Name resolution caching stores resolved names for incremental compilation. If a module's dependencies have not changed, its name resolution results can be reused from the cache.

Visibility checking enforces access control during name resolution. Private names are visible only within their defining module. Public names are visible from any module. Protected names are visible from the defining module and its descendants.

8.1 Target-Specific Code Generation

x86-64 code generation uses the System V ABI for function calls. Integer arguments are passed in registers (rdi, rsi, rdx, rcx, r8, r9) and return values in rax. Floating-point arguments use xmm registers.

AArch64 code generation uses the AAPCS64 calling convention. Arguments are passed in registers x0-x7 for integers and d0-d7 for floating-point. The link register (x30) stores the return address.

RISC-V code generation uses the standard RISC-V calling convention. Arguments are passed in registers a0-a7. The compiler generates compact instruction sequences using RISC-V's compressed (C) extension when available.

WebAssembly code generation produces Wasm bytecode for browser and server-side execution. The Wasm backend handles memory management through linear memory and implements the pipeline runtime as Wasm functions.

Calling convention adaptation for foreign function calls generates code that matches the target ABI. The compiler handles differences in argument passing, return value placement, and stack alignment.

Prologue and epilogue generation creates the function entry and exit code. The prologue saves callee-saved registers, allocates stack space, and sets up the frame pointer. The epilogue reverses these operations.

Exception handling code generation creates the unwind tables needed for stack unwinding. The tables describe the stack layout at each point in the function, enabling the unwinder to restore saved registers.

Debug info emission generates DWARF debug information for the target format. The debug info includes line number tables, variable locations, and type descriptions. Debug info enables source-level debugging.

Relocation generation creates entries for addresses that are not yet resolved. Relocations are resolved by the linker, which fills in the final addresses. Common relocation types include absolute, PC-relative, and GOT-relative.

Object file emission writes the generated code, data, and metadata to the target object file format (ELF, Mach-O, or COFF). The object file includes sections for code, data, BSS, debug info, and relocations.

7.2 Pipeline-Specific Optimizations

Iterator fusion recognizes iterator chains in the IR and merges them into a single loop. The fusion pass identifies the source, transformation stages, and sink of each pipeline. Fusible stages are merged into the loop body.

Collect elimination removes unnecessary materialization. A pipeline that collects into a vector and then immediately iterates over the vector is optimized to skip the collection entirely.

Filter-map fusion merges adjacent filter and map stages. The fused stage applies the filter predicate and the map function in a single iteration, avoiding the overhead of two separate function calls per element.

Fold specialization generates efficient code for common fold patterns. Summing, counting, and finding the minimum or maximum use specialized code that avoids the overhead of the general fold combinator.

Pipeline parallelization inserts parallel execution points in the pipeline. The optimizer identifies stages that can run concurrently and inserts work distribution and result collection code.

Sort optimization for pipelines recognizes sort-then-process patterns and selects the most efficient sorting algorithm based on the estimated element count and comparison cost.

Group-by optimization creates efficient hash-based grouping for group_by stages. The optimizer selects the hash function based on the key type and pre-sizes the hash map based on estimated group count.

Take optimization inserts early termination for pipelines with take(n) stages. The fused loop counts emitted elements and exits after n elements, avoiding processing the remaining source elements.

Flat-map fusion merges adjacent flat_map stages into a single nested loop. The optimizer determines whether the nested loop should be flattened (for small inner sequences) or kept nested (for large inner sequences).

Pipeline specialization creates type-specialized versions of generic pipeline stages. Specialization eliminates dynamic dispatch overhead and enables further type-specific optimizations.

5.2 Type Error Messages

Type error messages include the expected type, the actual type, the expression that caused the error, and the source location. Color-coded output highlights the relevant parts of the source code.

Suggestion generation proposes fixes for common type errors. Missing trait implementations suggest adding the implementation. Type mismatches suggest explicit conversion functions. Missing lifetimes suggest annotations.

Error chains show the sequence of type inference steps that led to the error. For pipeline type errors, the chain shows the type at each stage, highlighting the stage where the mismatch occurs.

Multiple error reporting continues type checking after the first error, reporting all errors in a single compilation. The type checker uses error recovery to produce meaningful diagnostics for dependent expressions.

Error deduplication removes redundant error messages. If the same type mismatch causes errors in multiple locations, only the first occurrence is reported, with a note about the number of suppressed duplicates.

Trait bound error messages explain which trait is required, which type does not implement it, and where the requirement originates. If the trait is implemented for a different type, the message suggests the correct type.

Lifetime error messages show the conflicting lifetimes with source annotations. The annotations indicate where each lifetime begins and ends, and which borrow creates the conflict.

Pattern matching error messages list the missing patterns when a match expression is not exhaustive. The missing patterns are displayed as source code that can be added to make the match exhaustive.

Import error messages suggest similar names when a name is not found. The suggestion considers all modules in the dependency tree, not just the current module. The closest matching name by edit distance is suggested.

Recursive type error messages detect and report infinite types. A type like struct Node { next: Node } has infinite size. The error suggests using Box<Node> for heap allocation.

6.2 SSA Construction

SSA construction uses the algorithm of Cytron et al., which inserts phi nodes at dominance frontiers. A definition in block B requires phi nodes at each block in B's dominance frontier that is reachable from B.

Iterated dominance frontier computation efficiently identifies all phi node insertion points. The algorithm starts from each definition's block and iterates the dominance frontier computation until no new blocks are added.

Variable renaming assigns version numbers to each variable definition. Each use of a variable references the version that reaches it. Phi nodes select the correct version based on the incoming edge.

Pruned SSA avoids inserting phi nodes for variables that are not live at the merge point. Pruning reduces the number of phi nodes, improving compilation speed and enabling more optimizations.

SSA deconstruction converts out of SSA form before code generation. Phi nodes are replaced with copy instructions on the incoming edges. Copy coalescing eliminates unnecessary copies.

Sparse SSA represents the def-use chains as explicit edges in the IR graph. Sparse SSA enables efficient traversal from definitions to uses and from uses to definitions.

Memory SSA extends SSA to memory operations. Each memory write creates a new version of the memory state. Memory SSA enables precise alias analysis and dead store elimination.

Loop-closed SSA form ensures that definitions inside a loop are visible outside only through phi nodes at loop exits. Loop-closed form simplifies loop optimizations by making loop boundaries explicit.

Critical edge splitting inserts empty blocks on edges from blocks with multiple successors to blocks with multiple predecessors. Splitting eliminates critical edges that complicate phi node placement.

SSA verification checks that every use is dominated by its definition, every phi node has the correct number of operands, and every branch targets a valid block. Verification runs after each optimization pass.

3.2 Error Recovery in Parsing

Panic mode recovery skips tokens until a synchronization token is found. Synchronization tokens include semicolons, closing braces, and keywords that start new statements (let, fn, struct). Recovery minimizes cascading errors.

Token insertion: when a closing delimiter is missing, the parser inserts the expected token and reports the missing delimiter. Insertion prevents errors in the code following the missing delimiter.

Token deletion: when an unexpected token appears, the parser deletes it and reports the extra token. Deletion handles common typos like double semicolons or extra commas.

Error production rules define common error patterns and their corrections. An error production for a missing type annotation suggests adding the type after the colon. Error productions provide specific, actionable error messages.

Context-sensitive error messages use the current parsing state to produce relevant messages. An error inside a match expression says 'expected match arm' rather than 'expected expression'.

Error limit prevents the parser from reporting too many errors. After 20 errors, the parser stops and suggests fixing the first errors before continuing. The limit prevents overwhelming output from severely broken code.

Indentation-aware recovery uses indentation to infer block boundaries when braces are mismatched. If a closing brace is missing, the recovery heuristic uses the indentation level to guess where the block ends.

Recovery quality metrics measure the fraction of correct AST nodes after error recovery. The Lateralus parser achieves 95% correct nodes when a single error is present, enabling useful IDE features even in broken code.

Incremental parsing reuses AST nodes that have not changed. The incremental parser identifies the changed region, re-parses it, and patches the AST. Incremental parsing provides sub-millisecond parse times for single-character edits.

Fuzzing the parser with randomly generated token streams tests error recovery robustness. The fuzzer verifies that the parser does not crash, loop infinitely, or produce inconsistent AST structures on any input.

8.2 Linking and Loading

The linker combines object files into a final executable. The linker resolves symbols, applies relocations, and arranges sections in the output file. Lateralus uses a custom linker for fast link times.

Symbol resolution matches symbol references to definitions. Strong symbols (function definitions) override weak symbols (default implementations). Undefined symbols produce link errors.

Section merging combines corresponding sections from different object files. Code sections are merged into a single executable segment. Data sections are merged into readable-writable segments.

Dead section elimination removes sections that are not referenced by any live section. Starting from the entry point and exported symbols, the linker traces references to identify live sections.

Dynamic linking generates shared libraries (.so files) with position-independent code. The dynamic linker resolves symbols at load time using the PLT (Procedure Linkage Table) and GOT (Global Offset Table).

Link-time garbage collection removes unused functions and data. The collector starts from exported symbols and traces all references, removing unreachable definitions. This reduces binary size significantly.

ASLR support generates position-independent executables that can be loaded at random addresses. Address randomization prevents memory layout prediction attacks.

Thread-local storage generates efficient TLS access patterns for thread-local variables. The linker selects the optimal TLS model (local-exec, initial-exec, or general-dynamic) based on the variable's scope.

Build ID generation embeds a unique identifier in the binary for matching with debug info. The build ID is a hash of the binary content, ensuring that debug info corresponds to the correct binary version.

Incremental linking updates only the changed sections of the binary. Incremental linking provides sub-second link times for small changes to large projects, significantly improving the edit-compile-test cycle.

2.2 Token Stream Processing

Token buffering collects tokens into a sliding window for lookahead. The parser can peek at upcoming tokens without consuming them. The buffer automatically refills from the lexer when tokens are consumed.

Token filtering removes whitespace and comment tokens from the stream before parsing. The filter preserves doc comment tokens for documentation generation. The filter runs lazily, processing tokens on demand.

Token position tracking maintains the current line and column for error reporting. The tracker accounts for tab stops, multi-byte Unicode characters, and newline variations (LF, CR, CRLF).

Token classification categorizes tokens for the parser's disambiguation logic. The classifier distinguishes keywords from identifiers, distinguishes unary from binary operators based on context, and handles context-sensitive tokens.

Macro expansion runs between lexing and parsing. The macro expander processes macro invocations, substituting arguments and expanding the macro body. The expanded tokens are fed back into the token stream for parsing.

Token tree construction groups tokens by matching delimiters (parentheses, brackets, braces). The tree structure enables macro processing that respects delimiter nesting without full parsing.

Conditional compilation processes cfg attributes during tokenization. Tokens within a disabled cfg block are skipped entirely, including their syntactic validation. This enables platform-specific code without parse errors.

Token stream serialization saves the token stream for incremental compilation. The serialized stream is loaded when the source file has not changed, avoiding re-lexing.

Token stream diffing compares old and new token streams to identify changed regions. The diff algorithm aligns tokens by position and content, identifying insertions, deletions, and modifications.

Token stream diagnostics check for common tokenization issues: mismatched delimiters, unterminated strings, and invalid escape sequences. These diagnostics are reported before parsing to provide early error detection.

5.3 Trait Resolution

Trait resolution determines which trait implementation to use for a method call. The resolver considers the receiver type, available implementations, and trait bounds on generic parameters.

Inherent method resolution checks for methods defined directly on the type (in an impl block without a trait). Inherent methods take precedence over trait methods with the same name.

Auto-deref resolution automatically dereferences the receiver to find a matching method. If x.method() does not match, the resolver tries (*x).method(), (**x).method(), and so on.

Auto-ref resolution automatically references the receiver to find a matching method. If x.method() does not match for a value receiver, the resolver tries (&x).method() and (&mut x).method().

Trait bound propagation passes trait bounds from function signatures to the bodies of generic functions. Within the body, the bounds are used to resolve trait method calls on generic parameters.

Associated type resolution determines the concrete type of associated types in trait implementations. The resolver uses the implementing type and trait to look up the associated type definition.

Blanket implementation resolution handles trait implementations that apply to all types satisfying a bound. Blanket implementations have lower priority than specific implementations.

Coherence checking verifies that there are no overlapping trait implementations. Two

implementations overlap if there exists a type that satisfies the bounds of both. Overlapping implementations are a compile-time error.

Specialization resolution selects the most specific implementation when multiple implementations are available. A more specific implementation has tighter bounds. Specialization enables performance optimization for specific types.

Trait resolution caching stores resolved trait implementations for reuse. The cache key is the combination of the trait, type, and generic arguments. Caching avoids redundant resolution in large codebases.

7.3 Register Allocation

Linear scan register allocation processes live intervals in order of their start positions. Each interval is assigned a register or spilled to the stack. Linear scan runs in O(n log n) time and produces good code for fast compilation.

Graph coloring register allocation models register assignment as a graph coloring problem. Nodes are live ranges, edges connect interfering ranges, and colors are registers. The allocator finds a coloring with minimum spill.

Live range splitting breaks long live ranges into shorter ones, reducing spill weight. Splitting at loop boundaries is particularly effective because it isolates loop-resident values from inter-loop values.

Spill code insertion adds load and store instructions for spilled values. The spiller minimizes the number of loads by rematerializing values that can be cheaply recomputed.

Coalescing eliminates unnecessary copy instructions by assigning the same register to the source and destination of a copy. Aggressive coalescing tries all copies, while conservative coalescing avoids increasing spill.

Caller-saved and callee-saved register management: the allocator preferentially assigns callee-saved registers to values that are live across calls. Caller-saved registers are used for values that die at calls.

Register pressure analysis estimates the number of simultaneously live values at each program point. High pressure indicates likely spilling. The optimizer uses pressure to guide decisions like inlining (which can increase pressure).

Phi node handling during register allocation assigns registers to phi inputs and outputs. Parallel copies at block boundaries resolve phi operands. The copies are coalesced when possible.

Stack layout assigns stack slots to spilled values and local variables. The layout respects alignment requirements and groups related values for cache locality.

Allocation verification checks that no two simultaneously live values share the same register, that all uses are preceded by definitions, and that calling conventions are respected.

9.1 Compiler Performance

Compilation speed averages 100,000 lines per second for debug builds and 30,000 lines per second for optimized builds. The difference is due to the optimization passes, which dominate compilation time for optimized builds.

Memory usage during compilation averages 50 bytes per source line. A 100,000-line project uses approximately 5 MB of compiler memory. The memory is dominated by the AST and IR representations.

Incremental compilation reduces rebuild time by 80-95% for small changes. The incremental compiler tracks dependencies at the function level, recompiling only functions that depend on changed definitions.

Parallel compilation distributes module compilation across CPU cores. A 4-core system compiles approximately 3.5x faster than a single-core system. The speedup is limited by dependencies between modules.

Compilation cache stores compiled artifacts for unchanged modules. The cache key includes the source hash, compiler version, and optimization level. Cache hits skip all compilation phases.

Batch compilation mode optimizes for throughput by processing all modules before starting optimization. This mode uses less memory than parallel mode because it can release AST memory after lowering each module.

Streaming compilation processes each module through all phases before starting the next module. Streaming mode minimizes peak memory usage and is preferred for memory-constrained systems.

Compiler benchmark suite measures compilation speed, memory usage, and generated code quality for a set of representative programs. The benchmarks run on every commit to detect performance regressions.

Generated code quality is measured by comparing execution time against hand-optimized C code. The Lateralus compiler generates code within 5% of hand-optimized C for compute-bound benchmarks.

Binary size for a hello-world program is 8 KB (statically linked) and 4 KB (dynamically linked). Binary size scales linearly with source code size, averaging 10 bytes of machine code per line of source.

4.2 Module System

The module system organizes code into hierarchical namespaces. Each source file defines a module. Directories define module hierarchies. The module tree is rooted at the crate (compilation unit).

Module visibility controls access to items. Public items (pub) are visible from outside the module. Private items (default) are visible only within the module. Pub(crate) items are visible within the crate.

Re-exports (pub use) make items from other modules available under a different path. Re-exports

simplify the public API by hiding internal module structure.

Circular dependency detection identifies and reports cycles in the module dependency graph. Circular dependencies are a compilation error because they prevent topological ordering of compilation units.

External crate resolution locates and loads dependency crates. The resolver reads the manifest file (lateralus.toml) for dependency declarations and searches the package registry for matching versions.

Module path resolution maps use declarations to module files. The declaration use std::collections::HashMap maps to the file std/collections.lat and the item HashMap within that file.

Conditional module inclusion uses cfg attributes to include modules only for specific platforms. A module with #[cfg(target_os = linux)] is compiled only when targeting Linux.

Module interface generation extracts the public API from a compiled module. The interface includes public types, functions, traits, and their signatures. The interface is used for type checking dependent modules.

Module documentation generation extracts doc comments and public signatures to produce HTML documentation. The documentation generator supports cross-references between modules and inline code examples.

Module testing compiles and runs test functions within each module. Tests are annotated with #[test] and executed by the test runner. Test results include pass/fail status and execution time.

6.3 Pipeline IR

Pipeline expressions in the IR are represented as a chain of closure applications on an iterator source. Each closure takes one element and produces zero or more elements. The chain is a first-class IR construct.

Pipeline IR nodes include: Source (the initial data), Map (element transformation), Filter (element selection), FlatMap (one-to-many transformation), Fold (accumulation), Collect (materialization), and Take (truncation).

Pipeline type information in the IR records the element type at each stage. The type information enables type-specialized code generation and optimization. Generic pipeline stages carry their instantiation types.

Pipeline cost annotations estimate the per-element processing cost of each stage. The cost model considers the number of instructions, memory accesses, and branch predictions. Cost annotations guide optimization decisions.

Pipeline fusion eligibility is marked on each stage. Eligible stages can be fused with adjacent stages. Ineligible stages (those with side effects or complex control flow) prevent fusion across them.

Pipeline parallelism annotations mark stages that can execute in parallel. The parallelism analysis considers data dependencies, effect ordering, and thread safety requirements.

Pipeline source analysis determines the source's element count, ordering, and uniqueness properties. These properties enable optimizations: known counts enable pre-allocation, known ordering enables sort elimination.

Pipeline sink analysis determines the sink's capacity and materialization requirements. The analysis enables pre-allocation for collect sinks and early termination for take sinks.

Pipeline IR pretty-printing produces a human-readable representation of the pipeline structure. The output shows each stage with its type, cost, and fusion eligibility.

Pipeline IR validation checks that stage types are compatible, fusion decisions are consistent, and parallelism annotations are safe. Validation runs after each pipeline optimization pass.

3.3 Precedence and Associativity

Operator precedence determines the binding strength of binary operators. Higher-precedence operators bind tighter. The Lateralus precedence table follows C conventions with additions for the pipeline operator.

The pipeline operator |> has precedence 1 (lowest), below assignment (2). This means that x = a |> f is parsed as x = (a |> f), not (x = a) |> f. The low precedence enables writing pipelines without parentheses.

Left associativity means that a |> b |> c is parsed as (a |> b) |> c. Most arithmetic operators are left-associative. The assignment operator is right-associative: a = b = c is parsed as a = (b = c).

Comparison operators are non-associative: a < b < c is a syntax error. This prevents the common mistake of writing chained comparisons that do not have the intended mathematical meaning.

Unary operator precedence is higher than all binary operators. Unary minus, logical not, and dereference bind tighter than any binary operation.

The Pratt parser implements precedence by passing the current precedence level to the expression parser. The parser continues consuming operators only while their precedence exceeds the current level.

Custom operator precedence allows user-defined operators to specify their precedence and associativity. The precedence declaration is part of the operator definition and is checked for consistency.

Precedence ambiguity detection identifies expressions that could be parsed differently with different precedence rules. The parser warns about potentially ambiguous expressions and suggests parentheses.

Mixed-mode expressions combine pipeline and arithmetic operations. The expression a + b |> f(c * d)

is parsed as (a + b) |> f(c * d), respecting the precedence hierarchy.

The precedence table is documented in the language reference with examples of each precedence level. The documentation includes a precedence diagram showing all operators and their relative binding strengths.

References

[1] Appel, A. Modern Compiler Implementation in ML. Cambridge University Press, 1998.

[2] Cooper, K. and Torczon, L. Engineering a Compiler, 2nd Ed. Morgan Kaufmann, 2011.

[3] Aho, A. et al. Compilers: Principles, Techniques, and Tools, 2nd Ed. Pearson, 2006.

[4] Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[5] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis. CGO, 2004.