From Zero to a Working Language

The Lateralus bootstrapping story: from Python prototype to self-hosted compiler

Lateralus Language

bad-antics · February 2026 · Lateralus Language Research

ABSTRACT Building a self-hosting compiler — one that compiles itself — is the traditional milestone for a new language. Lateralus achieved self-hosting in v0.5, after four bootstrap stages: a Python prototype, a C99 implementation, a partial Lateralus rewrite, and finally a fully self-hosted Lateralus compiler. This paper tells the bootstrapping story, the design decisions that were made differently at each stage, and the lessons learned about language design in the process of implementing it.

1. Stage 0: The Python Prototype

The first Lateralus compiler was 800 lines of Python. It supported only a minimal subset of the language: integer arithmetic, let bindings, function calls, and the total pipeline operator. The output was C99 source code.

```
# Python prototype: parsing a pipeline expression
def parse_pipeline(tokens):
    left = parse_expr(tokens)
    while peek(tokens) == '|>':
        consume(tokens, '|>')
        right = parse_call(tokens)
        left = Pipeline(left, right, variant='total')
    return left
```

The prototype proved the pipeline model was viable and produced the first working Lateralus programs. Its output quality was poor (no optimization, verbose C output) but it validated the syntax and semantics quickly.

The prototype took 3 weeks to build. Its value was in the design experiments it enabled: the original pipeline syntax used -> instead of |>, and the error operator was called ?> before settling on |?>.

2. Stage 1: The C99 Implementation

Stage 1 rewrote the compiler in C99 with a complete front-end (lexer, recursive descent parser, type checker) and an LLVM-based backend. This took 4 months and produced a compiler capable of bootstrapping itself.

The C99 compiler added: the full four-variant pipeline operator set, the row-polymorphic type system, the standard library (a subset sufficient to build the compiler), and the RISC-V backend (in addition to the x86-64 backend inherited from the Python prototype).

```
// C99 compiler: AST node for pipeline (abbreviated)
typedef struct {
    AstKind kind;       // AST_PIPELINE
    AstNode *left;      // left-hand expression
    AstNode *right;     // stage function
    PipeVariant variant; // TOTAL, ERROR, ASYNC, FANOUT
    SourceLoc loc;
} AstPipeline;
```

The C99 compiler passed all tests written for the Python prototype plus an additional 200 tests written for new language features.

3. Stage 2: Partial Lateralus Rewrite

The partial rewrite replaced the C99 front-end with Lateralus code while keeping the LLVM backend in C99. This was motivated by the discovery that the C99 type checker had several bugs in the row-polymorphism handling that were easier to fix in Lateralus than in C99.

```
// Lateralus type checker (replaces C99 equivalent)
fn infer_pipeline(env: &TypeEnv, left: &Ast, right: &Ast,
                  variant: PipeVariant) -> Result<Type, TypeError> {
    let left_type = infer(env, left)?;
    let right_type = infer(env, right)?;
    match variant {
        PipeVariant::Total =>
            unify(left_type, right_type.input)?;
            Ok(right_type.output)
        PipeVariant::Error =>
            // row-polymorphic Result handling...
    }
}
```

The partial rewrite uncovered 12 additional bugs, all in the error-propagation and row-polymorphism interaction. These were fixed in Stage 2's type checker and later backported to the C99 stage for historical reference.

4. Stage 3: Full Self-Hosting

Stage 3 replaced the C99 backend with a Lateralus backend. This required implementing register allocation, instruction selection, and the ABI conventions in Lateralus. The self-hosting milestone was reached when the Stage 3 compiler could compile the Stage 3 compiler and produce a binary that passed all tests.

```
// Lateralus register allocator: linear scan (excerpt)
fn allocate_registers(func: &IrFunction) -> RegAllocation {
    let intervals = compute_live_intervals(func);
    let mut active: Vec<LiveInterval> = Vec::new();
    let mut free_regs = CALLER_SAVED_REGS.to_vec();
    for interval in intervals.sorted_by_start() {
        expire_old(&mut active, &mut free_regs, interval.start);
        if free_regs.is_empty() {
            spill_at_interval(&mut active, interval)
        } else {
            assign_reg(interval, free_regs.pop().unwrap())
        }
    }
}
```

5. Lessons Learned: Language Design

The bootstrapping process revealed several language design issues that were not apparent from the specification alone:

- **Error type annotations are necessary**: the Stage 1 type checker inferred error types automatically, but this produced unreadable type errors when two stages returned different error types. Adding explicit error type annotations at pipeline boundaries made the errors local and actionable.

- **The row variable must be named**: initially, row variables were anonymous. When two anonymous row variables appeared in the same error message, the message was incomprehensible. Named row variables (r1, r2) made error messages parseable.
- **Constant-time mode must be opt-in**: the first version of constant-time mode was always-on. This prevented optimizations in non-crypto code. Making it an annotation (#[ct]) was the correct choice.

6. Lessons Learned: Compiler Architecture

The compiler architecture evolved significantly across the four stages:

- **Parse to typed AST in one pass**: the Python prototype had a separate parse and type-check pass. The C99 compiler merged them for performance. The Lateralus compiler separated them again for modularity. Separation won: it enables better error recovery and incremental compilation.
- **Pipeline IR nodes are essential**: Stage 1 desugared pipelines in the parser. This made the type checker and optimizer blind to pipeline structure. Stage 2 preserved pipeline nodes through to the optimizer, enabling fusion.
- **The back-end should not know about types**: the C99 backend had type-specific code generation scattered throughout. The Lateralus backend delegates all type decisions to the IR lowering pass and operates on untyped LIR.

7. The Self-Hosting Test

The definitive self-hosting test: compile the compiler, use the output to compile the compiler again, and verify that the two binaries are bit-identical:

```
# Self-hosting verification script
ltl build --release compiler.lt -o compiler_v1
compiler_v1 build --release compiler.lt -o compiler_v2
compiler_v2 build --release compiler.lt -o compiler_v3
diff compiler_v2 compiler_v3
# No output: v2 and v3 are identical (converged)
```

The v1-to-v2 transition may produce a different binary because the compiler being compiled has changed. v2 to v3 must be identical: if the compiler is correct, it produces the same output for the same input regardless of which generation compiled it.

8. Current State and Future

Lateralus v1.0 is self-hosted: the compiler is written in Lateralus, compiled by the Lateralus compiler, and passes the self-hosting test. The compiler codebase is 28,000 lines of Lateralus (including the standard library, but excluding tests).

Future work on the compiler: a JIT tier for the REPL (using the LBC interpreter as a profiling tier that feeds hot functions to a LLVM-based JIT), incremental compilation with fine-grained dependency tracking, and a parallel compilation mode that type-checks independent modules in parallel.