Bootstrapping a Compiler in Python

Stage 0 of the Lateralus compiler: lexer, parser, and bytecode emitter in 2000 lines

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT The first Lateralus compiler was written in Python. This was a deliberate choice: Python's expressiveness allowed rapid iteration on the language design, and the Python compiler served as the reference implementation against which the self-hosted Lateralus compiler is validated. This paper describes the architecture of the Python bootstrap compiler: its lexer, recursive-descent parser, type checker, and bytecode emitter. It also covers the validation process used to confirm correctness parity between the Python and Lateralus implementations.

1. Why Bootstrap in Python?

Language bootstrapping requires a working compiler before the language can compile itself. Common choices are C, an existing functional language, or a scripting language. We chose Python for three reasons:

- **Rapid iteration**: Python's dynamic typing and interactive shell allow the language design to be changed and tested in minutes rather than hours.
- **Readable AST manipulation**: Python's dataclasses and match statement (PEP 634) produce readable recursive tree walkers that mirror the grammar directly.
- **Testability**: Python's rich testing ecosystem (pytest, hypothesis) allows property-based testing of the compiler phases.

The Python bootstrap compiler (Stage 0) compiles a subset of Lateralus sufficient to implement the self-hosted compiler (Stage 1). Stage 0 is not shipped to users; it exists only to bootstrap the toolchain.

2. Compiler Architecture

The Stage 0 compiler has five phases:

```
source text
    → Lexer        (tokens)
    → Parser       (CST)
    → Resolver     (scope-resolved AST)
    → TypeChecker  (typed AST)
    → Emitter      (LBC bytecode)
```

Each phase is implemented as a class with a single entry point. The output of each phase is a Python dataclass tree that the next phase consumes. There is no shared mutable state between phases.

3. The Lexer

The lexer is a hand-written DFA that produces a flat token list. Python's string methods are used for character classification:

```
class Lexer:
    def __init__(self, src: str):
        self.src = src; self.pos = 0; self.tokens = []

    def lex(self) -> list[Token]:
        while self.pos < len(self.src):
            c = self.src[self.pos]
            if c.isspace():    self.skip_whitespace()
            elif c == '#':     self.skip_line_comment()
            elif c.isdigit():  self.lex_number()
```

```
            elif c.isalpha() or c == '_': self.lex_ident()
            elif c == '|':     self.lex_pipe_operator()
            else:              self.lex_symbol()
        return self.tokens
```

The lex_pipe_operator method handles the four pipeline operators: |>, |?>>, |>>, and |>|. Disambiguation requires two-character lookahead.

4. The Parser

The parser is a recursive-descent parser. Each grammar production is a method. Pipeline expressions have a dedicated production that left-folds operator-stage pairs into a tree:

```
def parse_pipeline(self) -> Expr:
    lhs = self.parse_application()
    while self.peek() in PIPE_OPS:
        op  = self.consume()
        rhs = self.parse_application()
        lhs = PipeExpr(op=op, lhs=lhs, rhs=rhs)
    return lhs
```

The resulting AST represents a |> f |> g as Pipe(Pipe(a, f), g) — a left-associative binary tree. This matches the operational semantics: left-to-right data flow.

5. Type Checking

The type checker performs constraint-based Hindley-Milner inference. Each AST node produces a constraint; the constraint set is unified by Robinson's unification algorithm:

```
def infer(self, node: Expr, env: TypeEnv) -> tuple[Type, Constraints]:
    match node:
        case Var(name):
            return env[name], []
        case PipeExpr(op=Total(), lhs=lhs, rhs=rhs):
            t_lhs, c_lhs = self.infer(lhs, env)
            t_rhs, c_rhs = self.infer(rhs, env)
            t_out = self.fresh_tvar()
            return t_out, c_lhs + c_rhs + [(t_rhs, Fun(t_lhs, t_out))]
        case _:
            raise TypeError(f'unhandled: {node}')
```

6. Bytecode Emission

The emitter walks the typed AST and produces LBC bytecode. Pipeline expressions emit a CALL instruction for each stage:

```
def emit(self, node: Expr, dst: Reg) -> None:
    match node:
        case PipeExpr(op=Total(), lhs=lhs, rhs=rhs):
            src = self.alloc_reg()
            self.emit(lhs, src)
            fn_reg = self.alloc_reg()
            self.emit(rhs, fn_reg)
            self.emit_instr(CALL, dst, fn_reg, src)
            self.free_reg(src); self.free_reg(fn_reg)
```

7. Validation: Differential Testing

The Stage 0 (Python) and Stage 1 (Lateralus) compilers are validated against each other via differential testing. A corpus of 8,000 Lateralus programs is compiled by both compilers; the LBC output is compared instruction-by-instruction:

```
# Differential test runner
for program in corpus:
    lbc_py  = stage0_compile(program)
    lbc_ltl = stage1_compile(program)
    if lbc_py != lbc_ltl:
        report_divergence(program, lbc_py, lbc_ltl)
```

Divergences in the differential test reveal bugs in one or both compilers. The Python compiler is considered the reference for the semantic subset it covers; Stage 1 bugs are fixed to match Stage 0.

8. Lessons Learned

Key insights from the Python bootstrap:

- Python's match statement (PEP 634) dramatically reduces the verbosity of recursive AST walkers — pattern matching is essential for compiler writers.
- Keeping phases stateless (each phase takes a tree, returns a tree) eliminates a class of ordering bugs and makes testing trivial.
- Differential testing is the most effective validation strategy during bootstrapping — unit tests miss subtle semantic divergences that differential testing catches immediately.
- The Python compiler remains valuable after Stage 1 ships: it is used as a fast alternative for tooling (IDE plugins, linters) that do not need the full compiler pipeline.