From Lexer to Language

Building the Lateralus compiler front-end: lexer, parser, name resolver, and type checker

Lateralus Language

bad-antics · October 2024 · Lateralus Language Research

ABSTRACT The Lateralus compiler front-end transforms source text into a fully type-checked intermediate representation in four stages: lexing, parsing, name resolution, and type inference. Each stage is implemented as a Lateralus pipeline stage, making the compiler a demonstration of its own language. This paper describes each stage's design, the data structures it produces, the errors it can report, and the performance characteristics that allow the compiler to process 100,000 lines per second on a single core.

1. Overview of the Front-End Pipeline

The compiler front-end is itself a pipeline:

```
source_text: &str
    |>  lex               // &str -> Vec<Token>
    |?> parse             // Vec<Token> -> Ast
    |?> resolve_names     // Ast -> NamedAst
    |?> infer_types       // NamedAst -> TypedAst
    |?> check_pipelines   // TypedAst -> TypedAst (pipeline variant check)
    |>  lower_to_ir       // TypedAst -> Ir
```

Each stage is a pure function except for name resolution, which reads the module import graph. Errors from any stage are collected and reported together at the end of the failed stage; subsequent stages do not run on a failed input.

2. Lexer: Token Stream Production

The lexer is a hand-written DFA operating on a byte slice. It produces a flat Vec<Token> where each token carries: kind, start byte offset, end byte offset, and (for string and numeric literals) a pre-computed value.

```
struct Token {
    kind:  TokenKind,
    start: u32,
    end:   u32,
}
```

Source locations are stored as byte offsets, not line/column pairs. Line/column is computed on demand from the offset using a precomputed newline index. This saves memory (2 × u32 vs 4 × u32) and avoids redundant computation for tokens whose location is never reported in an error message.

Throughput: 180 MB/s on typical Lateralus source files.

3. Parser: Recursive Descent with Pratt Expressions

The parser is a hand-written recursive descent parser for statements and declarations, with a Pratt parser for expressions. Pratt parsing handles the four pipeline operators and binary/unary arithmetic without special-casing precedence rules in the recursive descent.

3.1 Pratt Parsing the Pipeline Operators

Each pipeline operator is registered in the Pratt table with: a left-binding power (precedence), a right-binding power (associativity), and a handler function that constructs the AST node:

```
// Pratt table entry for |>
PrattEntry {
```

```
    token:     PIPELINE_TOTAL,
    lbp:       60,   // left-binding power
    rbp:       61,   // right-binding power (left-associative)
    nud:       None, // not a prefix operator
    led:       |left, ctx| Ast::Pipeline(left, PipelineKind::Total,
                                         ctx.parse_expr(rbp)),
}
```

The Pratt parser naturally handles mixed-variant pipelines because each operator has a separate table entry. No grammar rule needs to enumerate all operator variants; the dispatch is table-driven.

3.2 Error Recovery

The parser recovers from syntax errors by synchronizing on statement boundaries: when a parse error occurs, the parser discards tokens until it finds a ;, fn, let, or } and resumes parsing. This allows the compiler to report multiple independent syntax errors in one pass.

4. Name Resolution

Name resolution assigns a unique definition ID to every use of a name. It produces a NamedAst identical in structure to the raw Ast but with all identifier nodes replaced by DefId values.

The resolver walks the AST in two passes: a first pass that registers all top-level definitions (functions, types, module exports), and a second pass that resolves use-site identifiers against the scope chain built in the first pass.

4.1 Module Import Resolution

Module imports are resolved by following the module path from the workspace root. The resolver queries the module graph (a pre-built DAG of import relationships) to find the definition file, then calls the first-pass resolver on that file recursively. Circular imports are detected by tracking in-progress resolutions.

```
// Import resolution: path → module file → definition set
import std::data::Vec   // resolves to stdlib/data/vec.lt
import my_crate::utils  // resolves to src/utils.lt
```

5. Type Inference

The type checker uses Algorithm W with extensions for row polymorphism and the four pipeline operator variants. It produces a TypedAst where every expression node carries its inferred type.

The inference runs in a single bottom-up pass over the named AST. For each expression, it generates type constraints and unifies them. If unification fails, the error is reported with the origin locations of the conflicting types (as described in the error messages paper).

5.1 Pipeline Type Inference

Pipeline expressions are typed left-to-right. For each stage, the checker unifies the stage's expected input type with the output type of the previous stage. The operator variant determines the expected return type: total stages require B, error stages require Result<B, E>, async stages require Future<B>.

```
// Type inference trace for a mixed-variant pipeline
input: &[u8]
    |>  parse       // infers: (&[u8]) -> ParsedDoc
    |?> validate    // infers: (ParsedDoc) -> Result<ValidDoc, Error>
    |>> serialize   // infers: (ValidDoc) -> Future<Vec<u8>>
// final type: Future<Vec<u8>> (since last stage is async)
```

6. Pipeline Variant Checking

After type inference, the pipeline variant checker verifies that each operator variant is used correctly:

- |>: the stage function must return a plain type, not a Result or Future.
- |?>: the stage must return Result<B, E> and the error type must match the surrounding pipeline's error type.
- |>>: the stage must return Future<B>.
- |>|: the stage list must be non-empty and all stages must accept the same input type.

These checks are separate from type inference because they enforce semantic conventions that the type system alone cannot express: a function of type (A) -> Result<B, E> is type-valid as a |> stage (it just wraps its return in a Result), but it is semantically wrong — the programmer should use |?> to propagate the error.

7. Performance

The front-end processes approximately 100,000 lines per second on a single core of a 2024-era laptop. The bottleneck is type inference (60% of front-end time), followed by parsing (25%) and name resolution (15%). The lexer is negligible.

```
Stage            Time/KLOC  Memory/KLOC
-----------------------------------------
Lex                0.5 ms      2 KB
Parse              2.5 ms      8 KB
Name resolution    1.5 ms      4 KB
Type inference     6.0 ms     16 KB
Pipeline checks    0.5 ms      1 KB
Total             11.0 ms     31 KB
```

Incremental compilation reuses the type-checked AST for unchanged modules, reducing the effective cost to the changed files and their dependents.

8. Future Work

Planned improvements to the front-end: a lazy parser that skips function bodies not transitively imported by the current compilation unit (reducing parse time by 40-60% for large workspaces), a demand-driven type inference algorithm that avoids re-inferring types for unchanged expressions in incremental mode, and a language server protocol integration that reuses the front-end's incremental infrastructure for IDE hover and completion.