Lexer Design for a Pipeline-First Language

Tokenizing Lateralus: operator precedence, contextual keywords, and pipeline disambiguation

Lateralus Language

bad-antics · August 2024 · Lateralus Language Research

ABSTRACT The Lateralus lexer must handle four multi-character pipeline operators (|>, |?>, |>>, |>|), contextual keywords that are identifiers in some positions, and the disambiguation of | as either a bitwise-OR operator or the start of a pipeline operator. This paper describes the lexer's design: a hand-written DFA with a one-character lookahead, the contextual keyword table, and the two disambiguation rules. We discuss the trade-offs versus a generated lexer and explain why a hand-written DFA produces better error messages.

1. The Lexer's Context in the Compiler

The Lateralus compiler pipeline is: source text → tokens → AST → typed IR → optimized IR → machine code. The lexer is the first stage; it takes a byte stream and produces a token stream. The parser consumes the token stream and produces an AST.

The lexer's design decisions propagate forward: a poor tokenization choice forces the parser to carry disambiguating state, which complicates the grammar and produces worse error messages. The design principle is to resolve as much ambiguity as possible in the lexer, keeping the parser grammar context-free.

2. Operator Disambiguation

The character | appears in three roles in Lateralus:

- Bitwise OR: a | b
- Pattern match arm separator: Ok(x) | Err(x) => ...
- Start of a pipeline operator: |>, |?>, |>>, |>|

The lexer disambiguates by one-character lookahead: if the character after | is >, ?, or a second |, it enters pipeline-operator mode. Otherwise it emits a PIPE token (used for both bitwise OR and pattern separators; the parser distinguishes these by context).

```
// Lexer DFA: states for the | character
state AFTER_PIPE:
    '>'  -> emit PIPELINE_TOTAL,  return to INITIAL
    '?'  -> enter AFTER_PIPE_QUESTION
    '|'  -> enter AFTER_PIPE_PIPE
    else -> emit BITWISE_OR, re-process current char

state AFTER_PIPE_QUESTION:
    '>'  -> emit PIPELINE_ERROR,  return to INITIAL
    else -> error: expected '>' after '|?'
```

3. Pipeline Operator Token Table

The four pipeline operators tokenize to distinct token types with distinct precedence values baked in at lex time:

```
Source   Token             Precedence  Category
-------------------------------------------------
|>       PIPELINE_TOTAL    6           total
|?>      PIPELINE_ERROR    6           error
|>>      PIPELINE_ASYNC    6           async
|>|      PIPELINE_FANOUT   6           fan-out
```

All four share precedence 6 (lower than arithmetic, higher than comparison operators). The shared precedence means no parentheses are needed in the common case of mixing operator variants in one expression. The precedence was chosen empirically by testing 200 real Lateralus programs and measuring how often parentheses were needed under various precedence assignments.

4. Contextual Keywords

Lateralus has a small set of contextual keywords: identifiers that are reserved in specific syntactic positions but legal as identifiers elsewhere. The contextual keywords are:

```
Keyword    Contextual position
------------------------------
pipe       After '=' or '(' at expression start
async      Before 'fn' or '|>'
sealed     Before 'record' or 'enum'
pub        Before 'fn', 'let', 'record', 'enum', 'module'
where      After a type parameter list
with       After 'import'
```

The lexer emits these as identifier tokens (IDENT) and the parser promotes them to keyword tokens based on position. This avoids breaking code that uses these words as variable names, which is common in practice (let pipe = ... appears in tests and teaching material).

5. The Hand-Written DFA Rationale

Lexer generators (flex, ANTLR's lexer mode) produce correct lexers automatically from a grammar specification. Why write the Lateralus lexer by hand?

Three reasons: error message quality, contextual keyword support, and compile-time predictability.

5.1 Error Message Quality

A generated DFA reports 'unexpected character' with the character and position. A hand-written DFA can report the context: 'expected > after |? to form the error pipeline operator'. This context-aware message is possible only because the hand-written code knows it is in AFTER_PIPE_QUESTION state.

5.2 Contextual Keyword Support

Contextual keywords require state that a simple DFA does not carry. The hand-written lexer threads a 'previous token' value through the tokenization loop, allowing the contextual keyword table lookup to check whether the preceding token was an assignment operator, a parenthesis, or a declaration keyword.

5.3 Compile-Time Predictability

The generated DFA size and performance depend on the lexer generator's output, which can vary between versions. The hand-written DFA has a fixed, reviewable implementation that does not change when the tool is updated.

6. Unicode Handling

Lateralus source files are UTF-8. The lexer handles Unicode in two phases: identifier scanning and string literal scanning.

Identifier scanning accepts the Unicode XID_Start and XID_Continue character classes as defined by Unicode 15.0. Operator characters are ASCII-only; the four pipeline operators use only |, >, and ?.

String literals accept any valid UTF-8 byte sequence. Escape sequences follow Rust conventions: \u{XXXX} for Unicode code points up to U+10FFFF, \n, \t, \\, and \".

```
// Valid Lateralus identifiers
let café = "espresso"
let π = 3.14159
let ■■ = "Taro"

// Invalid: operator characters in identifiers
let x|y = 1  // error: | is an operator, not an identifier char
```

7. Numeric Literal Lexing

Numeric literals support four bases and two floating-point formats:

```
// Integer literals
255        // decimal
0xFF       // hexadecimal
0b11111111 // binary
0o377      // octal
1_000_000  // underscore separators allowed

// Floating-point literals
3.14
1.0e-10
0x1.8p+0   // hex float (IEEE 754 hex format)
```

The lexer distinguishes integer and float literals without looking at the suffix: any literal containing a . or e/E exponent is a float. Hex floats use p/P as the exponent marker (IEEE 754 convention).

8. Benchmark: Lexer Throughput

We measured the Lateralus lexer throughput on three representative inputs: the full Lateralus standard library source (82K tokens), a generated stress test with maximum pipeline operator density (100K pipeline operators in 1M tokens), and a Unicode-heavy source with mixed-script identifiers.

```
Input                     Size     Throughput
-----------------------------------------------
Standard library          82K tok   180 MB/s
Pipeline stress test      1M tok    165 MB/s
Unicode-heavy source      500K tok  140 MB/s
```

The pipeline stress test is 9% slower than the standard library input because the four-character pipeline operators require two lookahead reads each (the first | and then the operator suffix). The Unicode overhead reflects the XID character class table lookup. All three inputs are well within the threshold for interactive compilation (target: > 50 MB/s).

Lateralus is an open-source, zero-dependency programming language. Project home: https://lateralus.dev. Source: github.com/bad-antics/lateralus-lang. Released under CC BY 4.0.