Writing a REPL: Interactive Development in Lateralus

bad-antics | March 2024 | Tutorial

Abstract

This paper documents the design and implementation of the Lateralus REPL, an interactive development environment for the Lateralus programming language. We cover the architecture of the REPL's incremental lexer, parser, and evaluator, the line editor with syntax highlighting and multi-line support, the tab-completion engine with type-directed suggestions, and the pipeline visualization feature. The REPL maintains full compatibility with the batch compiler while providing instant feedback for exploratory programming.

1 Introduction

A Read-Eval-Print Loop is the primary interactive interface for exploring a programming language. For Lateralus, the REPL serves as both a learning tool and a rapid prototyping environment, allowing developers to test pipeline expressions, inspect types, and explore the standard library without compiling a full program.

This paper documents the architecture and implementation of the Lateralus REPL, covering the line editor, incremental lexer, incremental parser, expression evaluator, error presentation layer, and tab-completion engine. Each subsystem is designed to provide instant feedback while maintaining full compatibility with the batch compiler's semantics.

The REPL is implemented in approximately 4,200 lines of Lateralus code and links against the compiler's front-end libraries. It supports multi-line editing, syntax highlighting in the terminal, and persistent history across sessions.

2 Architecture

The REPL follows a layered architecture where each stage communicates through well-defined interfaces. At the outermost layer, the line editor captures keystrokes and manages the terminal buffer. Completed lines are fed into the incremental lexer, which produces a token stream that is forwarded to the incremental parser.

The parser produces an AST fragment, which is then type-checked against the accumulated environment. If type-checking succeeds, the AST is lowered to MIR and evaluated by a tree-walking interpreter. Results are pretty-printed with type annotations back to the terminal.

```
  struct ReplState {
      env: Environment,
      history: Vec<String>,
      line_editor: LineEditor,
      lexer_state: LexerCheckpoint,
      parser_state: ParserCheckpoint,
```

```
      type_ctx: TypeContext,
      eval_ctx: EvalContext,
      completions: CompletionEngine,
  }
```

Error recovery is critical: when any stage fails, the REPL must report a helpful diagnostic and return to a clean state rather than crashing. We implement this with a checkpoint-restore mechanism at each pipeline stage.

3 Line Editor

The built-in line editor provides GNU Readline-compatible keybindings with extensions for Lateralus-specific features. It supports Emacs and Vi editing modes, configurable via a dotfile at ~/.lateralus/repl.toml.

Multi-line input is detected by tracking open delimiters: parentheses, brackets, braces, and pipeline continuations. When the parser signals an incomplete expression, the editor switches to continuation mode, displaying a secondary prompt and indenting automatically.

```
  fn handle_key(key: Key, state: &mut EditState) -> Action {
      match key {
          Key::Char(c) => state.insert(c),
          Key::Tab => trigger_completion(state),
          Key::Enter => {
              if state.is_balanced() {
                  Action::Submit(state.drain())
              } else {
                  state.newline_continue()
              }
          }
          Key::CtrlC => Action::Abort,
          Key::CtrlD => Action::Exit,
          _ => state.handle_motion(key),
      }
  }
```

Syntax highlighting is performed in real-time as the user types. The lexer runs on each keystroke to produce token spans, which are mapped to ANSI color codes. Keywords are displayed in bold cyan, strings in green, numbers in yellow, and pipeline operators in magenta.

4 Incremental Lexing

Rather than re-lexing the entire input on every keystroke, the REPL maintains a lexer checkpoint that records the lexer's automaton state at the end of the last successfully lexed prefix. When the user edits the buffer, only the changed region and subsequent tokens are re-lexed.

The incremental lexer handles Lateralus-specific tokens including the pipeline operator |>, the

reverse pipeline <|, lambda arrows =>, pattern-match arms ->, and string interpolations delimited by ${}. Each of these multi-character tokens requires careful handling at edit boundaries.

```
fn incremental_lex(buf: &str, cp: &LexerCheckpoint) -> TokenStream {
    let start = cp.byte_offset;
    let mut lexer = Lexer::resume(cp);
    let mut tokens = cp.prior_tokens.clone();
    tokens.truncate(cp.valid_count);
    for tok in lexer.lex(&buf[start..]) {
        tokens.push(tok);
    }
    TokenStream::new(tokens)
}
```

5 Incremental Parsing

The parser consumes the token stream and builds an AST. In REPL mode, the parser must distinguish between incomplete input (which should trigger continuation) and genuinely erroneous input. We solve this by annotating parse errors with an 'incomplete' flag that indicates whether additional tokens could make the expression valid.

Expression parsing follows the Pratt parsing algorithm with precedence climbing. Pipeline operators |> and <| are parsed as binary operators at precedence level 1, just above assignment. This means x |> f |> g parses as (x |> f) |> g, giving the natural left-to-right data flow.

```
fn parse_expr(p: &mut Parser, min_prec: u8) -> Result<Expr> {
    let mut lhs = parse_atom(p)?;
    loop {
        let op = match p.peek() {
            Some(tok) if is_binop(tok) => tok,
            _ => break,
        };
        let (lp, rp) = prec_of(op);
        if lp < min_prec { break; }
        p.advance();
        let rhs = parse_expr(p, rp)?;
        lhs = Expr::BinOp(op, Box::new(lhs), Box::new(rhs));
    }
    Ok(lhs)
}
```

6 Expression Evaluation

Once an expression is parsed and type-checked, it is lowered to MIR and evaluated by an interpreter. The interpreter maintains a value environment that persists across REPL inputs, allowing let-bindings and function definitions to accumulate over a session.

Pipeline expressions are evaluated lazily when possible. The expression xs |> filter(is_even) |> map(square) |> take(5) constructs a lazy iterator chain that only materializes values on demand. The REPL pretty-printer shows the first few elements and indicates remaining count.

```
fn eval_pipeline(stages: &[PipeStage], env: &mut Env) -> Value {
    let mut val = eval_expr(&stages[0].expr, env);
    for stage in &stages[1..] {
        val = match &stage.kind {
            PipeKind::Call(f) => {
                let func = env.lookup(f);
                func.apply(val)
            }
            PipeKind::Lambda(params, body) => {
                let mut inner = env.child();
                inner.bind(&params[0], val);
                eval_expr(body, &mut inner)
            }
        };
    }
    val
}
```

7 Error Presentation

Errors in the REPL are displayed with the same diagnostic quality as the batch compiler. Each error includes a source span, a primary message, optional secondary labels, and a suggested fix when available. Colors follow the convention: red for errors, yellow for warnings, and blue for informational notes.

When a type error occurs in a pipeline, the diagnostic highlights the exact stage where the type mismatch happens and shows the expected and actual types in a side-by-side display. This is implemented by threading source-location metadata through every MIR node.

8 Tab Completion

The completion engine provides context-sensitive suggestions based on the current cursor position and the accumulated environment. At the top level, it suggests let-bound names, imported modules, and keywords. After a dot, it suggests fields and methods of the inferred type.

Completion results are ranked by relevance using a scoring function that considers edit distance, frequency of use in the current session, and type compatibility. The top 15 results are displayed in a scrollable popup rendered with ANSI escape sequences.

```
fn complete(buf: &str, pos: usize, env: &Env) -> Vec<Completion> {
    let prefix = extract_prefix(buf, pos);
    let ctx = infer_context(buf, pos, env);
    let mut candidates = match ctx {
```

```
            Ctx::TopLevel => env.all_names(),
            Ctx::DotAccess(ty) => ty.fields_and_methods(),
            Ctx::Import => available_modules(),
            Ctx::PipeRhs(ty) => fns_accepting(ty, env),
        };
        candidates.retain(|c| c.name.starts_with(&prefix));
        candidates.sort_by(|a, b| score(b).cmp(&score(a)));
        candidates.truncate(15);
        candidates
    }
```

9 Pipeline Visualization

A unique feature of the Lateralus REPL is the :pipeline command, which renders a visual representation of the data flow through a pipeline expression. Each stage is displayed as a box with its input and output types, connected by arrows.

The visualization uses Unicode box-drawing characters and is rendered directly in the terminal. For complex pipelines with branching (using the tee operator), the display shows a tree structure. This feature has proven invaluable for debugging long pipeline chains in real-world Lateralus code.

```
// Example REPL session:
lat> :pipeline data |> parse_csv |> filter(valid) |> group_by(.region)

  [Vec<u8>] --> |parse_csv| --> [Vec<Record>]
                                      |
                                 |filter(valid)|
                                      |
                                [Vec<Record>]
                                      |
                             |group_by(.region)|
                                      |
                          [Map<String, Vec<Record>>]
```

10 Conclusion

The Lateralus REPL demonstrates that interactive development environments can maintain the same semantic rigor as batch compilers while providing instant feedback. Key design decisions include incremental processing at every stage, checkpoint-based error recovery, and context-sensitive completion that understands the type system.

Future work includes adding a debugger integration that allows setting breakpoints in REPL-defined functions, a notebook mode for literate programming, and WebAssembly compilation for running the REPL in a browser.

3.1 History Management

The REPL stores command history in ~/.lateralus/history, a plain-text file with timestamps. History search supports both prefix matching (up-arrow) and substring matching (Ctrl-R). Duplicate consecutive entries are automatically de-duplicated.

Each history entry records the wall-clock time, the number of tokens, and whether the expression evaluated successfully. This metadata enables the :history --errors command, which shows only entries that produced diagnostics, useful for reviewing past mistakes.

```
struct HistoryEntry {
    timestamp: Instant,
    input: String,
    token_count: usize,
    success: bool,
    result_type: Option<Type>,
}

fn save_history(path: &Path, entries: &[HistoryEntry]) {
    let file = File::create(path).unwrap();
    for e in entries {
        writeln!(file, "{}|{}|{}", e.timestamp, e.input, e.success);
    }
}
```

The history file format is intentionally simple--one entry per line with pipe-delimited fields--so it can be processed by standard Unix tools. Entries containing newlines use a continuation marker (backslash-newline) to preserve multi-line inputs.

History completion integrates with the tab-completion engine: when the user presses Ctrl-R, the REPL switches to history search mode and displays matches inline. Pressing Ctrl-R again cycles through older matches, while Ctrl-S cycles forward.

- - Up/Down arrows: navigate history sequentially
- - Ctrl-R: incremental reverse search
- - Ctrl-S: incremental forward search
- - Alt-.: insert last argument of previous command
- - :history N: show last N entries
- - :history --errors: show only failed entries

Session isolation is supported through the :session command, which creates a named history partition. This is useful when working on multiple projects simultaneously, as completion suggestions are scoped to the active session's history.

The REPL also supports persistent undo within a session. Each successfully evaluated expression is recorded as a transaction, and the :undo command rolls back the environment to the state before the last expression. This is implemented by snapshotting the environment's binding map at each step.

6.1 Value Representation

Values in the REPL interpreter are represented as a tagged union. Primitive types (integers, floats, booleans, characters) are stored inline, while compound types (strings, arrays, records) use reference-counted heap allocation to avoid deep copies during pipeline evaluation.

```
enum Value {
    Int(i64),
    Float(f64),
    Bool(bool),
    Char(char),
    Str(Rc<String>),
    Array(Rc<Vec<Value>>),
    Record(Rc<HashMap<String, Value>>),
    Func(Rc<Closure>),
    Unit,
}
```

The use of reference counting rather than garbage collection keeps REPL response times predictable. Cycle detection is unnecessary because Lateralus's ownership model prevents reference cycles in well-typed programs, and the REPL enforces the same type discipline.

Pretty-printing of values respects the terminal width and uses a layout algorithm inspired by Wadler-Lindig. Nested structures are displayed with indentation, and large collections are truncated with an ellipsis showing the total element count.

The REPL supports custom pretty-printers via the Display trait. User-defined types can implement Display to control how their values appear in REPL output. The default printer shows the constructor name and field values in a structured format.

```
fn pretty_print(val: &Value, width: usize) -> String {
    match val {
        Value::Array(arr) if arr.len() > 10 => {
            let head: Vec<_> = arr[..5].iter()
                .map(|v| format_val(v)).collect();
            let tail: Vec<_> = arr[arr.len()-2..].iter()
                .map(|v| format_val(v)).collect();
            format!("[{}, ... ({} more), {}]",
                head.join(", "), arr.len() - 7, tail.join(", "))
        }
        _ => format_val(val),
    }
}
```

Closures capture their environment by value, following Lateralus's move semantics. When a closure is created in the REPL, the captured bindings are cloned into the closure's environment. This ensures that subsequent REPL inputs cannot invalidate the closure's captured state.

The interpreter also supports special REPL-only values such as the implicit result variable _, which holds the value of the last evaluated expression. This mirrors the convention found in many interactive environments and allows quick chaining of exploratory computations.

References

[1] Flatt, M. 'Creating Languages in Racket.' Communications of the ACM, 2012.

[2] Kery, M.B. et al. 'The Story in the Notebook.' CHI 2018.

[3] Ford, B. 'Parsing Expression Grammars.' POPL 2004.

[4] Pratt, V. 'Top Down Operator Precedence.' POPL 1973.

[5] Heeren, B. et al. 'Scripting the Type Inference Process.' ICFP 2003.

[6] Barras, B. et al. 'The Coq Proof Assistant Reference Manual.' INRIA, 2010.

[7] Klabnik, S. and Nichols, C. 'The Rust Programming Language.' No Starch Press, 2019.