Writing a REPL for Lateralus

Incremental compilation, persistent scope, and pipeline evaluation in interactive mode

Lateralus Language

bad-antics · January 2025 · Lateralus Language Research

ABSTRACT A Read-Eval-Print Loop (REPL) for a statically-typed, compiled language presents challenges that interpreted languages avoid: types must be inferred for incomplete expressions, bindings from previous entries must persist in scope, and the compiler must accept code fragments that would be invalid in a standalone file. The Lateralus REPL (ltl) solves these problems by maintaining a persistent compilation unit that grows with each entry, using a relaxed type inference mode for top-level expressions, and mapping pipeline evaluation to an interactive display format. This paper describes the architecture, the incremental compilation strategy, and the REPL's integration with the language server.

1. The Challenges of a Typed REPL

A REPL for a dynamically-typed language is straightforward: evaluate the expression, print the result, and update the environment. For a statically-typed language, three additional concerns arise:

- **Incomplete types**: the user may enter an expression whose type cannot be inferred from the expression alone, because the relevant type is defined in a future entry.
- **Persistent bindings**: a let declaration in entry N must be in scope for entries N+1, N+2, ... This requires the type environment to accumulate across entries.
- **Effectful expressions**: an expression that performs I/O or modifies state cannot simply be re-evaluated if the type checker needs to retry with additional context.

The Lateralus REPL addresses each concern with a specific design decision.

2. The Persistent Compilation Unit

Rather than treating each REPL entry as an independent program, the REPL maintains a single growing module — the persistent compilation unit (PCU). Each entry appends to the PCU and the PCU is re-type-checked incrementally.

```
// Entry 1: user types
let x = 42

// PCU after entry 1:
module __repl__ {
    let x: i32 = 42   // type inferred
}

// Entry 2: user types
let y = x * 2

// PCU after entry 2:
module __repl__ {
    let x: i32 = 42
    let y: i32 = x * 2  // x is in scope
}
```

The PCU is never compiled to a standalone binary; it exists only in the compiler's memory and is used as the type context for each new entry.

3. Incremental Type Checking

When a new entry is added, the REPL type-checks only the new declarations against the accumulated type environment. Existing declarations are not re-checked unless the new entry redefines a name (in which case the old definition is shadowed and its dependents are re-checked).

Name shadowing in the REPL follows a different rule from the language specification: in a file, shadowing a let binding is a lint warning; in the REPL, it is the normal way to redefine a value after correcting a mistake.

3.1 Relaxed Type Inference for Top-Level Expressions

In a REPL session, the user often types an expression (not a let binding) to inspect its value. The type checker accepts bare expressions at the top level of a REPL entry and infers their type. If the type is Result<T, E> or Future<T>, the REPL automatically unwraps the result for display and prints the error if the value is Err.

4. Pipeline Evaluation and Display

Pipeline expressions are a first-class concern in the REPL. When the user enters a pipeline, the REPL evaluates each stage and displays intermediate results:

```
// User types:
[1, 2, 3, 4, 5]
    |> filter(|x| x % 2 == 0)
    |> map(|x| x * x)

// REPL output (with --verbose-pipeline):
stage 0: [1, 2, 3, 4, 5]  (input)
stage 1: [2, 4]            (after filter)
stage 2: [4, 16]           (after map)
=> [4, 16]: Vec<i32>
```

The intermediate results are computed by the interpreter, which evaluates the pipeline stage-by-stage rather than fusing stages as the AOT compiler does. This is intentional: the programmer wants to see each intermediate value.

4.1 Error Display for |?>

If a |?> stage fails, the REPL shows the error at the failed stage and stops evaluation, mirroring the runtime behavior:

```
// User types:
"not a number" |?> parse_i32

// REPL output:
error at stage 1: ParseIntError { input: "not a number" }
=> Err(ParseIntError { ... }): Result<i32, ParseIntError>
```

5. The Interpreter vs Compiler Path

The REPL uses a bytecode interpreter for evaluation, not the AOT compiler. The interpreter operates on the same typed IR that the AOT compiler produces, but evaluates it directly rather than emitting machine code. This avoids the startup cost of machine code generation for short expressions.

```
Mode          Startup   Throughput      Use case
-------------------------------------------------
```

```
Interpreter   < 1 ms    ~10 M ops/s    REPL, tests
JIT (planned) ~5 ms     ~500 M ops/s   REPL hot paths
AOT           ~50 ms    native speed    production
```

For performance-sensitive REPL work (benchmarking a data structure, profiling an algorithm), the user can invoke the AOT compiler from the REPL with :compile and time the compiled version.

6. Language Server Integration

The REPL shares its type inference engine with the language server protocol (LSP) implementation. The LSP uses the same incremental type checker: as the user types in the editor, the LSP re-checks only the changed declarations and reports type errors inline.

The REPL and LSP share the PCU abstraction: the LSP's in-progress file is treated as a PCU that is re-checked on every keystroke. This means the REPL and the LSP have identical type checking behavior, so there are no 'the REPL accepts it but the compiler rejects it' surprises.

7. REPL Commands

The REPL provides a set of meta-commands prefixed with ::

```
:type <expr>      Show the inferred type of an expression
:doc <name>       Show documentation for a name
:reset            Clear the PCU and start fresh
:load <file>      Load a source file into the PCU
:compile <expr>   AOT-compile and time the expression
:pipeline <expr>  Show stage-by-stage pipeline evaluation
:history          Show previous entries
:quit             Exit the REPL
```

The :type command is particularly useful for understanding how the type system infers the output type of a pipeline without running the pipeline.

8. Limitations and Future Work

Current limitations of the Lateralus REPL: the interpreter does not support all language features (async pipelines require a runtime scheduler that the interpreter does not include), the PCU does not persist across REPL sessions (planned for v2.0 via a serializable type environment), and the verbose pipeline display does not show intermediate values for lazy iterators (they are forced to evaluate eagerly for display, which may diverge for infinite iterators).

Future work: a JIT compilation path for hot REPL code, session persistence, and an integration with the notebook interface (Lateralus notebooks are a planned feature that uses the REPL as its evaluation engine).