Wasome Compiler

Architecture Analysis & Critique

01 01, 1980

Introduction

This document provides a detailed analysis of the Wasome compiler's components and overall architecture. It begins with a high-level component overview, followed by an in-depth, "brutally honest" critique of the individual crates and the system's design.

Component Overview

The compiler is structured as a workspace with four main crates: `source`, `shared`, `lexer`, and `ast`.

- **`source` crate:** Responsible for loading and managing source code files. It features a `FileLoader` trait for abstracting file system access and a `SourceMap` to manage loaded files.
- **`lexer` crate:** Responsible for lexical analysis. It uses the `logos` library to convert source code into a stream of tokens.
- **`ast` crate:** Defines the Abstract Syntax Tree, the central data structure representing the code's structure.
- **`shared` crate:** A collection of common code, primarily for handling source code locations. Most of this crate is deprecated.

Architectural Diagrams

To better visualize the critique, here are two diagrams representing the **current** flawed architecture and a **suggested** improved architecture. *Note: These diagrams use built-in Typst features to ensure compilability without external packages.*

Current Architecture

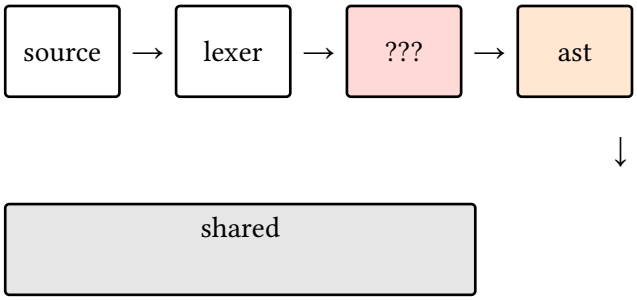

Figure 1: Current, flawed architecture. Note the missing Parser and the problematic responsibilities of the `ast` and `shared` crates.

Suggested Architecture

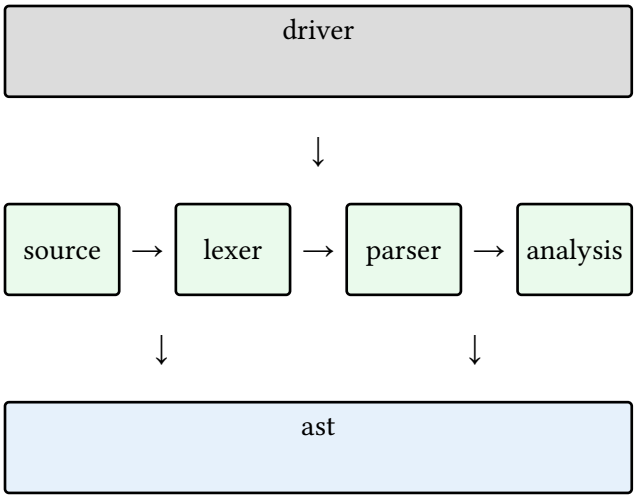

Figure 2: Suggested, cleaner architecture with clear separation of concerns and a central `driver` crate.

The `ast` Crate: A Honest Critique

⚠️ **Architectural Flaw** Yes, the `ast` crate is doing too much, and it's a significant architectural issue.

It Violates the Single Responsibility Principle (SRP)

The `ast` crate should have one responsibility: **to define the data structures that represent the code's syntax tree.**

Instead, it has absorbed the responsibilities of at least three different compiler components:

- **AST Definition:** The structs and enums for `Expression`, `Statement`, etc. (This is its correct job).
- **Semantic Analysis:** It actively validates code. The prime example is `FunctionCall::new` for the `TypedAST`, which performs type checking on function arguments. This is the job of a type-checker, not a data structure's constructor.
- **Symbol Table Management:** It defines `Symbol`, `FunctionSymbol`, and the `SymbolTable` trait. The symbol table is a distinct data structure that is central to semantic analysis, not to the raw AST.
- **Scope Resolution:** The `traversal` helpers are not generic visitors; they are specifically built to resolve symbols by walking up the tree. This is scope analysis logic that belongs in a semantic analysis pass.

The "Self-Typechecking" AST is a Design Flaw

The `Crates.md` file mentions the `TypedAST` is "self-typechecking". This is not a feature to be proud of; it's a design smell.

- **It Tightly Couples Data and Logic:** By embedding validation logic directly into the AST nodes, the data structure becomes inseparable from the language's specific rules.
- **It Obscures the Compiler's Flow:** A compiler has distinct phases. By having the AST "do things" on its own, it blurs the lines between these phases and makes the architecture harder to understand.

Other Components: A Honest Critique

`source` Crate: Good, But Not Bulletproof

✅ **Overall Design** The `source` crate is quite well-designed. The `SourceMap` concept is a cornerstone of a good compiler. The handling of multi-byte UTF-8 characters for column calculation is non-trivial and impressively thorough. The `FileLoader` trait shows good foresight for testing and abstraction.

⚠️ **Brutally Honest Critique**

1. **API Safety is Questionable:** The functions `lookup_location` and `get_source_slice` have documentation that basically says, "If you give this a `Span` from a different `SourceMap`, all bets are off." A truly robust API would not put this burden on the caller.
2. **Error Handling is Generic:** The methods return `std::io::Error`. This is lazy. A top-tier library would define its own, more descriptive `SourceError` enum.
3. **The `FileLoader` Abstraction is Leaky:** The default `WasomeLoader` uses `std::fs::canonicalize`, while the `MockLoader` in tests does not. This means the mock doesn't fully replicate the behavior of the real loader, creating a testing gap.

`lexer` Crate: Solid, But Lacks Pro-Grade Features

✅ **Foundation** The lexer works and is built on a solid foundation (`logos`), which is a great choice for performance and maintainability.

⚠️ **Missing Features**

1. **No Error Recovery:** The lexer is 'fail-fast'. It sees one bad token and gives up, which is a poor user experience. Production compilers can typically recover and report multiple errors at once.
2. **Character Literal Support is Incomplete:** The `char_callback` only supports a small, hardcoded list of escape sequences. It completely lacks support for Unicode (`\u{...}`) or hex (`\x...`) escapes.
3. **Identifier Definition is Too Permissive:** The regex `r"[a-zA-Z_][a-zA-Z0-9_]*"` completely disallows non-ASCII identifiers, a major limitation for a modern language.

`shared` Crate: A Messy, Unfinished Refactor

✅ **Intentions** The code within it is marked `#[deprecated]`. This shows someone knew it was wrong and had a plan to fix it.

⚠️ **Execution**

1. **It's a Zombie:** The crate is deprecated, but it's not dead. The `ast` crate **still uses it** (`ASTNode` stores a `CodeArea` from `shared`). This is an unforgivable mess. It means the refactoring to `source` was started but never finished.

2. **It's Actively Harmful to Development:** A new developer sees a `shared` crate and assumes it's important, creating confusion. It's a sign of poor code hygiene and incomplete work. The `shared` crate should have been **eliminated**.

Overall Architecture and Suggestions

- **Biggest Missing Piece:** The most significant architectural issue is the **absence of a parser or a "driver" crate**.
- **Top Suggestions:**
  1. **Aggressively Refactor the `ast` Crate:** This is the highest priority. Move all semantic analysis, symbol table, and traversal logic into a new, separate `analysis` or `semantics` crate.
  2. **Finish the `shared` Crate's Removal:** Update the `ast` crate to use `Span` from the `source` crate instead of `CodeArea` from `shared`. After that, **delete the `shared` crate**.
  3. **Implement a `parser` Crate:** This is the next logical step to make the compiler functional.
  4. **Create a `driver` Crate:** A top-level binary crate (e.g., `wasomec`) will be needed to tie everything together.

In conclusion: The foundation of the Wasome compiler is strong, but it suffers from a major architectural flaw in the `ast` crate and an incomplete, messy refactoring of the `shared` crate. Prioritizing a clean separation of concerns is critical for the long-term health of this project.