17-350 / 17-650

Lecture 02: Bytecode Interpreters and WebAssembly

Jonathan Aldrich
Aug 27, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Lecture Plan

- Cargo and the HW1 starter project
- Why tree-walking interpreters are slow
- Bytecode compilation
- Direct slot indexing
- Case study: WebAssembly bytecode

Next lecture: fast interpreters

Rust's Cargo

- Build tool + package manager + test runner, in one
- Replaces separately wiring together a compiler, a dependency manager, and a test harness
- Everything for HW1 goes through `cargo`, driven by `Cargo.toml`

Core Cargo Commands

- `cargo build` - compiles the project, output in `target/debug/...`
  - `cargo build --release` for production (faster & smaller code, no debug info)
- `cargo run [-- args...]` - build + run the binary
  - Interpreter demo: `cargo run -- test/add.snek`
  - HW1: `cargo run -- fib.wat fib 10`
- `cargo test` - runs `#[test]` functions and everything in `tests/`
  - passing HW1 == `tests/interpreter_test.rs` passing
- `cargo check` - fast compile-only check while iterating
- `cargo fmt` - reformats your code in a rust-standard way

`Cargo.toml`

```toml
[package]
name = "hw1-interpreter"
version = "0.1.0"
edition = "2021"

[dependencies]
sexp = "1.1.4"
```

- `[package]`: name, version, edition
- `[dependencies]`: crates from crates.io (here, `sexp` for S-expression parsing)
  - `sexp = "1.1.4"` allows any version of `sexp` below `2.0.0` and >= `1.1.4` (*Semantic versioning*)
- `[edition]`: what version of the Rust language to use. "2024" adds features (e.g. proper `async` closures) but can break certain "2021" code

`Cargo.lock`

- `Cargo.toml` states version *requirements*; `Cargo.lock` records the exact versions actually resolved
  - Cargo will pick the highest version number that satisfies compatibility
- Committed for binaries/applications -> reproducible builds
  - Don't commit for libraries, for client flexibility
- Regenerated automatically; you normally don't hand-edit it

Project Layout for HW1

- `src/lib.rs` - the library you implement (`Module`, `Function`, parsing, execution)
- `src/main.rs` - CLI driver, already provided
- `tests/` - Cargo's convention for integration tests, compiled against your public API
- `main.rs` can `use hw1_interpreter::Module` because the binary crate depends on its own library crate by package name

Demo: HW1 Starter Code Walkthrough

[Live walkthrough] Tour of src/lib.rs and main.rs

- `Result<T, String>` and `?` for error handling
- `#[derive(Debug, Clone, Default)]`
- `impl`: define methods associated with a type
- `HashMap<String, T>` for `funcs` / `exports`
- Pattern matching on `Sexp` / `Atom`
- `&str` vs `String` (why `atom_text` returns `Option<&str>`)
- What you still need to design: representation of bytecode in the `Function` struct

Are Interpreters Still Relevant?

- Yes: an interpreter is often the fastest way to make a language, format, or policy useful
  - A core part of Ben Titzer's research here at CMU
- **Rapid iteration:** change the language or behavior without rebuilding a native compiler
- **Portability:** run the same program across operating systems, devices, or sandboxes
- **Extensibility:** plugins, configuration languages, query engines, rules, and automation
- **Diagnostics:** easy to get context about language execution
- **Safety and isolation:** validate or restrict untrusted code before it reaches the host system
- Why care about speed?
  - Languages and tools may execute billions of operations
  - Fast startup and predictable latency matter for short-lived tools, servers, and interactive systems
  - A fast interpreter is a strong foundation for later specialization or JIT compilation

Fast, Safe Interpreters in Rust

- How do we write a fast interpreter?
  - Course theme: leverage Rust's low-level facilities make systems code fast
- For various aspects of an interpreter, ask:
  i. What does it cost in a naive tree-walker?
  ii. How much can we improve in purely **safe** Rust?
  iii. What more can be done using `unsafe` features (or bleeding edge nightly features) - and how do we contain any risk?
- HW1 is a WebAssembly interpreter
  - You can just write a simple tree-walking interpreter for HW1
  - But you will use these techniques to optimize it in HW2
    - Can you write the fastest interpreter in the class?

Discussion: Why is Tree-Walking Slow?

Think about the sample interpreter we saw in the last class

- What might be inefficient about the tree traversal/matching approach?
- Assume our `snek` language is extended with variables and operations `read x` and `write y = <value>`
  - what's the easiest way to implement this?
  - is there a way to make it faster with a little more work?

Why Tree-Walking Is Slow (Summary)

- `interpret_expr(&Expr)` walks boxed AST nodes:
  - Pointer chasing - each node is a separate heap allocation, poor cache locality
  - `match` must not only dispatch on the node type, but extract all the relevant information from the node data structure
  - Environment lookups by name on every variable reference
  - Redundant work: doing all of the above every time a function or loop body runs
- The AST is a bad in-memory representation for *repeated* execution

Bytecode Compilation

- Compile the AST once into a flat, linear sequence of instructions; interpret *that*
- Benefits:
  - `Vec<Instr>` instead of a tree -> cache-friendly, sequential access
  - Dispatch reads an opcode instead of re-matching a tree shape
  - Opens the door to peephole optimization, constant folding, superinstructions
- Safe-Rust path: define an instruction `enum`, store instructions in a `Vec`, and dispatch with `match`
  - The compiler checks that every instruction variant is handled
  - Gets most of the bytecode win with zero `unsafe`

A Safe Bytecode Dispatch Loop

```
#[derive(Clone, Copy)]
enum Instr { Const(i32), Add, Halt, }
fn run(code: &[Instr]) -> Result<i32, &'static str> {
        let mut stack = Vec::new();
        let mut pc = 0;
        loop {
                let instr = *code.get(pc).ok_or("program counter out of bounds")?;
                match instr {
                        Instr::Const(value) => stack.push(value),
                        Instr::Add => {
                                let rhs = stack.pop().ok_or("stack underflow")?;
                                let lhs = stack.pop().ok_or("stack underflow")?;
                                stack.push(lhs.checked_add(rhs).ok_or("integer overflow")?);
                        }
                        Instr::Halt => return stack.pop().ok_or("empty stack"),
                }
                pc += 1;
        }
}
```

- `code` is a flat, cache-friendly instruction array
- `pc` specifies the next instruction; `match` dispatches to its behavior
- Bounds checks and error handling remain explicit and safe

Bytecode: Design Choices

- Stack-based (push/pop operands) vs register-based (operands addressed by slot)
  - More on this trade-off in the next lecture
- `enum`-based instructions (safe, ergonomic, larger due to tag + payload) vs raw byte opcodes (compact, but needs unsafe/careful decoding)
- Discussion: what did we give up moving from a tree to a flat list?
  - Structured control flow and direct recursion -> now need explicit jump targets

Direct Slot Indexing

- Problem: name-based variable lookup costs something on *every* access
- Fix: resolve names to slot indices at compile time; runtime access is `locals[i]`
- Same idea as stack frames in compiled code: fixed offsets instead of symbolic lookup
- Fully safe: `Vec<Value>` locals + a compile-time resolution pass producing `usize` indices
- Bounds checks stay (safe); the *design* (arrays vs maps) is what buys the speed
- HW1 tie-in: `local.get $x` / `local.set $x` are named locals a real implementation resolves to indices

WebAssembly: Design Overview

- A real, heavily-optimized bytecode designed for exactly the properties we just discussed
- Stack machine at the bytecode level
  - Stack layout is statically known, so stack contents can be stored in registers in an optimized implementation, spilling to memory for very deep stacks
- Local variables via direct slot indices (`local.get` / `local.set`)
- It's also the exact format HW1 interprets

WebAssembly: Structured Control Flow

- `block` / `loop` / `if`; branches target enclosing structures by depth
- No arbitrary `goto` - contrast with "jump to arbitrary offset" bytecode
- Why: makes control flow provably reducible -> simpler, faster validation and compilation

WebAssembly: Folded WAT

```
(module
  (func $fib (export "fib") (param $n i32) (result i32)
    (if (result i32)
      (i32.lt_s (local.get $n) (i32.const 2))
      (then
        (local.get $n)
      )
      (else
        (i32.add
          (call $fib (i32.sub (local.get $n) (i32.const 1)))
          (call $fib (i32.sub (local.get $n) (i32.const 2)))
        )
      )
    )
  )
)
```

- From HW1's `tests/fib.wat`
- Each nested list is an instruction whose operands are subexpressions

From Folded WAT to Linear Instructions

```
local.get 0                 ;; n
i32.const 2
i32.lt_s
if result i32
        local.get 0             ;; n
else
        local.get 0             ;; n
        i32.const 1
        i32.sub
        call $fib
        local.get 0             ;; n
        i32.const 2
        i32.sub
        call $fib
        i32.add
end
return
```

- The tree's nesting becomes instruction order plus structured control flow
- The operand stack carries intermediate results between instructions
- The translation to linear bytecode can replace `$n` with slot `0` before execution
- This is a readable linear sketch, not the exact binary encoding

WebAssembly: Validation and Trade-offs

- Single-pass, linear type-checking before execution - no unbounded lookahead
- Enables fast startup and safety guarantees without per-op runtime type checks
- Determinism and portability constraints trade off some optimization opportunities for safety
- Wasm = "what you get when safety, speed, and provable validation are first-class design goals"
- Wasm runtimes themselves are written in Rust/C++, using many of the unsafe techniques from the next lecture

Summary

- We derived ideas like bytecode & slots, then saw how WebAssembly puts those ideas into practice
- HW1 asks you to implement an interpreter for WebAssembly
  - you don't have to do the step of moving to linear bytecode
  - but doing so probably gives you a head start for HW2
- Next lecture: Fast Interpreter Implementation
  - The interpreter hot loop: dispatch
  - Where safe Rust hits its limits, and what unsafe buys you
  - "Nightly" prototype Rust features that can achieve both safety and performance