17-350 / 17-650

Lecture 05: Practical Language Implementation

Jonathan Aldrich
Sep 8, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Today: Engineering Guidance for HW2

- General guidance
  - Static name resolution and validation
  - Incremental development, testing and debugging
- Direct Threading
  - ABI and stack obligations of handler-to-handler jumps
  - Compact code and resolved runtime references
- JIT Compilation
  - Compile-time environments and machine-code control flow
  - Function frames, calls, and runtime error boundaries

Resolving Names Before Execution

For efficiency, resolve statically (at "compile time", before interpreter or JIT code execution starts):

```
$local     -> local slot index
$function  -> function-table index
$label     -> threaded-code target
```

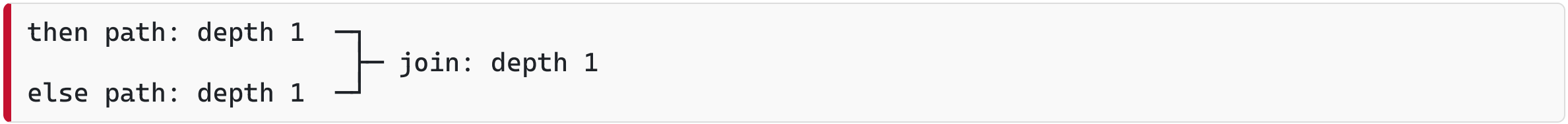
Avoids repeatedly hashing strings or searching nested labels at run time. If turns into a set of conditional and unconditional jump instructions (so the threaded interpreter executes a slightly different "bytecode" than raw WASM).

Control Flow Must Preserve Stack Shape

Track the operand stack depth at compile time:

```
i32.const       0 -> 1
i32.add         2 -> 1
local.set       1 -> 0
call f (n args)  n -> 1
```

At every control-flow join, all incoming paths must agree:



- Reject underflow before emitting unchecked pops
- Reject branches that arrive with an incompatible depth
- Check that a function reaches its end with exactly one result
- This proof supports safe execution and justifieds unchecked pop / unwraps (if you use them)

Develop your interpreter/JIT incrementally to aid debugging

1. Keep the tree-walking interpreter as an executable semantic oracle.
2. Implement and test constants, locals, and one binary operator.
3. Add conditionals.
4. Add function prologues, epilogues, and nonrecursive calls.
5. Add recursive calls (for JIT: check alignment and frame restoration).
6. Add locals, loops, and branches with stack-shape validation.
7. Benchmark (optional).

At each step, compare the compiled result with the interpreter on small programs before using the larger benchmarks.

Testing Your Implementation

As time permits, test boundary conditions, not just the benchmarks we gave you:

- nested arithmetic, subtraction, signed remainder, and wrapping overflow
- zero-, one-, and multi-argument calls
- recursion and nested calls
- local initialization, mutation, and unknown locals
- loops that execute zero, one, and many times
- nested blocks and loops with branches in both directions
- malformed input, unknown labels, wrong arity, and invalid stack shapes

Try both debug and release builds. Inspect generated assembly / threaded code when a result is wrong.

To save you time / complexity: **you do not have to catch divide by zero errors, nor the case of i32::MIN % -1.**

Recall the basic idea of Direct Threading

```
dst = load ip
ip  = ip + PTR_SIZE
jmp dst
```

A practical implementation raises some questions:

- How to represent WASM locals?
- How do we make jumps compatible with the Rust ABI?
- What happens to a Rust handler's stack frame?
- How do we return at the end?

Representing WASM Locals

- Lecture 3 assumed a `VM` but there wasn't much in it
  - In the sample code, just the WASM execution stack and a program counter
- We could enhance the VM with more information
  - A call stack, with an array of locals for each
- Alternatively, we can replace the VM with a (set of) `Frame`s
  - Each `Frame` holds the execution stack and locals for the current function, plus some VM information
  - Pass the Frame to `jump_to` instead of the VM
  - Executing a WASM `call` creates a new `Frame` and jumps to the function's instructions; when the function returns, we return at the Rust level and continue with the previous `Frame`.

Function prologues and epilogues

A normal Rust function has a prologue and epilogue:

```
push r14
sub  rsp, 32
; handler body
add  rsp, 32
pop  r14
ret
```

A direct `jmp` to a handler bypasses the epilogue.

- Stack space is not reclaimed
- Callee-saved registers may not be restored
- The final `ret` might return to an address given by some random word on the stack, rather than the one put there by the original `call`

Discussion: Function prologues and epilogues

In the direct threaded interpreter technique, a jump at the end of one handler to the next handler skips the epilogue for the first handler, so:

- Stack space is not reclaimed
- Callee-saved registers may not be restored
- The final ret might return to an address given by some random word on the stack, rather than the one put there by the original call

How can these challenges be addressed?

A prologue for the whole threaded execution trace

- Each handler might save different callee-save register values.
- We want the registers at the beginning of the whole sequence to be restored at the end, So we write a custom assembly "trampoline" to do so:

```
unsafe extern "sysv64" { fn threaded_run(vm: *mut c_void, entry: usize, ip: *const c_void) -> i32;}
global_asm!(
    r#"
    .global threaded_run
    .type threaded_run,@function
threaded_run:
    push rbx
    ...
    push r15
    ; save the stack pointer to the VM / Frame data structure
    ; load the first handler and jump to it
```

- At trampoline entry, the `VM` pointer will be in an argument register (e.g. first is `rdi`). If we declare `VM` as `#[repr(C)]` and put the stack pointer field first, then `mov [rdi], rsp` writes that field.
- The stored value is the stack position *after* saving the caller's callee-saved registers. It is the point to which every handler must return the stack.
- We invoke the handler by jumping to another argument register

Epilogue(s) for the threaded execution trace

We need an epilogue (or two epilogues, one for pass and one for fail) corresponding to the prologue from the previous slide:

- Takes the `VM` / `Frame` as an argument
- restores `rsp` from the VM / frame
- restores all the callee-save registers
- stores a value in `rax` indicating a successful halt or an execution error
- finally returns back to the Rust execution driver with `ret`

Reusing stack space on each jump

Each time we jump to a Rust handler, stack space is allocated. We want to reuse it.

- One way to do this is by extending `jump_to`:

```
#[cfg(target_arch = "x86_64")]

unsafe fn jump_to(next: Handler, vm: &mut VM, ip: *const Thread) -> ! {
    unsafe {
        asm!(
            "mov rsp, [rdi]",
            "jmp {target}",
            target = in(reg) next,
            in("rdi") vm as *mut VM,
            in("rsi") ip,
            options(noreturn)
        )
    }
}
```

- On each jump we should restore `rsp`
  - the `in("rdi") vm` declaration means `vm` is in `rdi`
  - this code assumes the stored stack pointer is in the first word of the `VM` data structure

Three Handler Designs

Design	Consequences
Rust handler + assembly jump	Pay cost of prologue/epilogue, may reload VM/IP on every jump
Naked assembly handler	Most streamlined option, but can't use standard Rust features
Tail calls with nightly Rust	Compiler handles streamlining; incorporate `unsafe` in some other way (HW2 requires some `unsafe`)

A naked function has no compiler-generated prologue or epilogue. Its body cannot contain ordinary Rust logic such as `Vec::push`, allocation, or `Result` construction:

```
#[unsafe(naked)]
unsafe extern "sysv64" fn op_add() -> ! {
    core::arch::naked_asm!("mov rax, [rdi] ..."
```

The second design is likely highest performance, because it omits the prologue/epilogue, but is very low level - could call into Rust for some opcodes though. The third is probably easiest.

Compact Threaded Code

A compromise between storing a large AST-like enum in every instruction cell and instructions as a list of handlers with instruction arguments somewhere separate.

```rust
#[derive(Debug, Clone)]
#[repr(C)]
struct Instruction {
    handler: Handler,
    operand: usize,
}
```

- `i32.const` : immediate is the constant
- `local.get` : immediate is a local-slot index
- `br` : immediate is a handler index
- `call` : immediate encodes a resolved target and arity
- `i32.add` : no immediate needed

Dense code improves cache locality and minimizes decoding overhead. Making the immediate optional (as in actual assembly or bytecode) would make the code even denser, but safety becomes more complex.

Function Calls vs. Threaded Handlers

Threaded dispatch eliminates returns; ordinary function calls return as usual.

Before a call:

- Evaluate arguments in the specified order
- Verify/precompute arity
- Respect System V stack alignment
- Ensure thread/VM state can be reloaded after return

Recursive language calls will still consume the native stack unless the language implementation uses an separate call stack.

The Cost of Embedding in Rust

Direct threading is not automatically faster. Potential costs can hide the dispatch win:

- Rust handler prologues and register spills
- Restoring stack state before every jump
- `Vec` growth or temporary argument allocation
- Dynamic name lookup
- Large instruction cells and cache misses

Suggested Development Order

1. Implement a correct safe bytecode interpreter.
2. Compile names and labels to numeric indices.
3. Add compact decoded operations and stack validation.
4. Benchmark before adding `unsafe` (optional, but interesting).
5. Encapsulate the unsafe dispatcher behind a private type.
6. Test debug **and** release builds.

Think about correctness and a specific safety argument when first adding `unsafe`, not as an afterthought.

Safety Argument Checklist

- **Hazards:** invalid code pointer, IP, frame pointer, stack state, ABI state
- **Invariant:** precise conditions on code cells, targets, stack depths, and registers
- **Establish:** parser/compiler/trampoline establish conditions before dispatch
- **Maintain:** every handler and exit path preserves or restores conditions

Add a `// SAFETY:` explanation at each unsafe boundary. Test malformed programs as well as valid benchmarks.

Alternative: JIT Compilation

Homework 2 offers two different ways to make execution faster:

Threaded interpreter	JIT compiler
Store program as a list of handlers and dispach	Emit native instructions for the program
Optimize the interpreter's execution loop	Replace the execution loop with compiled code
Validate a representation executed repeatedly	Validate the code before making it callable

The approaches share front-end work:

- Parse the WAT code
- Resolve names and function arities
- Preserve WebAssembly evaluation order and `i32` semantics
- Reject malformed programs before unsafe execution

From Evaluation to Code Generation

Start with the tree-walking interpreter from HW1:

```
evaluate(expression, environment) -> i32
```

Keep the recursive structure but process differently at each node

```
compile(expression, environment) -> machine instructions
```

- `i32.const 5` evaluates to `5`; compilation emits `mov eax, 5`
- `local.get $n` reads a map entry; compilation emits a load from a fixed offset
- `i32.add` computes now; compilation emits code that computes later
- The generated code leaves its result in `eax` (in our convention)

The compiler splits the work into two phases: syntax and names are handled at JIT compile time. values are computed when the native code runs.

Compile-Time Environments

Run-time name lookup becomes compile-time layout:

```
$n      -> parameter slot 0 -> [r12 + 0]
$i      -> local slot 1     -> [r13 + 8]
$sum    -> local slot 2     -> [r13 + 16]
```

- Compute the parameter/local layout before emitting the body
- Reject duplicate names and unknown locals during compilation
- Keep function signatures for checking `call` arity
- Allocate labels for branches as lexical scopes are entered

The generated code contains offsets and labels, not strings or hashmap lookups.

The JIT Frame

The JIT can use the same frame convention for every compiled function. This example extends the one from Lecture 4 for locals, using `r13` as the base register.

```
entry:
    push r12              ; save caller's parameter base
    push r13              ; save caller's local base
    mov  r12, rdi         ; incoming pointer to parameters
    sub  rsp, local_bytes
    mov  r13, rsp         ; space for this function's locals
    ; ... generated body, result in eax ...
    add  rsp, local_bytes
    pop  r13
    pop  r12
    ret
```

- Parameters and locals use fixed-width slots for simple addressing
- Locals are initialized before the body runs
- `r12` and `r13` are callee-saved, so recursive calls can restore each caller's frame
- Every path that reaches the epilogue must leave a valid `i32` result in `eax`

Two Stacks, One `rsp`

The machine stack stores several forms of data:

- return addresses from `call`
- saved registers and local storage
- temporary values from nested expressions (the WASM stack)
- arguments for WASM-level calls

Make sure things match up - for every emitted `push`, ask:

1. What value is being preserved?
2. Which later `pop` or stack adjustment removes it?
3. Does this change the alignment at the next native `call`?

The WebAssembly operand stack is nested within each stack frame of the machine stack.

Lowering `block`, `loop`, and Branches

The lecture 4 `if` pattern generalizes to labels:

```
(loop $again
  ...
  (br_if $again condition)
)
```

```
again:
    ; ... body ...
    ; condition in eax
    test eax, eax
    jnz again
after_loop:
```

- A branch to a `loop` targets the loop's beginning
- A branch to a `block` targets the instruction after the block
- Resolve labels while compiling, and scope nested labels correctly

Compiling Calls

Language calls become ordinary native calls in the minimal JIT:

```
; compile arguments, pushing them so memory is in source order
mov rdi, rsp            ; callee receives pointer to argument slots
sub rsp, 8              ; if required for System V alignment
call function_label
add rsp, argument_bytes + padding
```

- Check the target exists and the argument count matches before code generation
- Evaluate and push arguments left to right (as specified by WASM)
- Respect the ABI in both the caller and callee
- Native recursion uses the native call stack

Alignment is an ABI obligation, not a WebAssembly semantic rule.

Runtime Errors and the Unsafe Boundary

A tree interpreter can return `Result` from every recursive evaluation. Generated machine code usually has a plain `i32` return convention.

Ideally reject before execution:

- malformed syntax and unsupported instructions
- unknown locals, labels, and functions
- wrong call arity
- inconsistent operand-stack shapes, stack underflow
- invalid entry points or missing exports

Then check what remains at runtime, e.g. divide by zero

The safe public API is responsible for ensuring invalid programs execute as unchecked machine code.

JIT Safety Argument

Use the same Hazard-Invariant-Establish-Maintain structure as for threading.

- **Hazards:** wrong function signature, invalid entry or branch target, non-executable memory, bad stack alignment, invalid local offsets, corrupted frame state
- **Invariant:** every recorded entry and label identifies generated code in the executable buffer; every function follows the same ABI and frame layout; every generated path has a valid stack shape
- **Establish:** parsing, name resolution, arity checks, stack validation, label resolution, and assembler finalization establish the invariant before invocation
- **Maintain:** prologues and epilogues restore callee-saved state; calls preserve the frame convention; the executable buffer outlives all derived function pointers

`mem::transmute` does not check any of this. It only makes the compiler trust the invariant you established.

Advice on 4 HW2 paths

- JIT: relatively straightforward, provides best speedups for longer-running code (maybe all code)
- Direct threading with nightly: longer but clear path, as the core implementation technique is safe.
  - Use `unsafe` as required by the homework to avoid checks that are not needed due to bytecode validation.
- Direct threading with assembly: more complex, two options
  - Rust handlers w/ assembly jumps: take advantage of Rust conveniences, but have to watch ABI
  - Naked assembly handlers: probably fastest threading approach, but write more in assembly and may still have to call into rust for more sophisticated features

Summary

- General guidance
  - Do static name resolution and validation first
  - Practice incremental development, testing and debugging
- Direct Threading - we discussed:
  - ABI and stack obligations of handler-to-handler jumps
  - Compact code and resolved runtime references
- JIT Compilation - we discussed:
  - Compile-time environments and machine-code control flow
  - Function frames, calls, and runtime error boundaries

Some HW2 advice:

- The JIT approach is probably the shortest/easiest solution, despite what you might think
- Direct threaded with unsafe jumps is interesting but probably hardest due to managing the ABI
- You CAN do direct threaded with nightly. The threading is safe but you could use `unsafe` to eliminate run-time checks that you know are safe because of validation. A little longer than the JIT approach but not really harder.
- Naked assembly handlers are probably the fastest direct threaded approach (haven't prototyped)