

Ruby Wasm Compiler

Mark Murphy

Advisor: Brian Ladd
Semester: Fall 2024

August 25, 2025

Abstract

The Computer Science capstone proposal is a 500 -1000 word essay that addresses the topics described below. This template is intended to provide guidance regarding the structure and content of the proposal. The proposal itself must be written in paragraph/narrative format. Generally, both senior projects and internships require the same information in the proposal. Where appropriate, the template explicitly addresses differences between information required for senior projects and internships. Please see the *Computer Science Major's Capstone Requirements* document for more detailed information and explanation of the ideas discussed in this template.

1 Introduction

I propose to create an Ahead of Time (AoT) compiler for an as-yet-unspecified subset of the Ruby programming language. The only compilation target will be WebAssembly 3.0, the newest version of WebAssembly, with support for garbage-collected memory [1].

WebAssembly (Wasm) is a virtual machine (VM) specification, similar to the Java Virtual Machine or the .NET Common Language Runtime, although lower-level. Its most notable feature is that every major web browser includes a Wasm VM, allowing it to be used as a non-Javascript compilation target for languages targeting the browser.

While WebAssembly has had browser support since 2017, the initial 1.0 release supported only manual memory management. So, shipping Wasm 1.0 code for any garbage-collected language also requires shipping a garbage collector, which runs separately from the browser's built-in garbage collector. In contrast, Wasm 3.0 includes the WasmGC proposal [2], currently supported in all major browsers [3]. WasmGC introduces struct and array types managed by the VM [4].

With these changes, a compiler targeting Wasm is a much less heavy lift, closer to a bytecode interpreter than a full compiler with backend. There are WasmGC ports for Java, Kotlin, and OCaml. Of particular interest is Hoot, a compiler for Guile Scheme targeting WasmGC [5] [6]. Similarly to Ruby, Scheme is a dynamic, metaprogramming-heavy language, so this should be a good source.

There exist several projects which compile a Ruby interpreter to Wasm [7] [8]. However, to my knowledge there exists no project that attempts to compile Ruby code directly to WasmGC. This novelty is one reason I'm interested in Ruby in particular. I also think it's a charmingly quirky language. I understand Ruby to be somewhat more popular in Japan, its country of origin, which is a good fit for my future ambitions.

I don't know yet exactly what from Ruby to include and exclude. Certain core features (File, IO, Thread) are impossible within the browser, or would need to be adapted. I would be pretty happy to have a good approximation of Ruby's object and class facilities, and the normal data structures of a high-level programming language. This will include dealing with strings (Javascript interoper and the JS String Builtins Wasm proposal seem promising), interned symbols, dynamically-sized lists, and a hash-table implementation, and function closures. I don't expect I'll be able to implement any large portion of the standard library, or successfully compile any existing non-trivial Ruby programs.

I intend to write the compiler in Rust, because it's my favorite programming language. It's also a pretty convenient language for writing interpreters/compilers, since it has both ML-like algebraic data types and nice byte-manipulation facilities.

2 Deliverables

The main deliverable of this project is a compiler binary, which takes in files of the Ruby subset, and produces Wasm binaries. The compiler should also be able to .wat files, Wasm's textual format. This will be important for debugging, and should be fairly trivial given an otherwise-working compiler. As discussed, the compiler will be written in Rust.

A complete (4.0-deserving) project should also include a small demonstration project, which runs in the browser. A very simple todo application would be good, I think. It can demonstrate Javascript interop and DOM manipulation. That will consist of a ruby-subset file, a small HTML file, and a Javascript scaffolding file, which loads and runs the Wasm file, and provides the required imports.

I would consider the project's documentation to be a deliverable, as well as the capstone presentation slides. In addition to normal internal API documentation and instructions for installing and running the compiler, I'd like to include fairly in-depth architectural documentation, explaining the compilation pipeline and discussing technical choices made.

Along with the project, I intend to produce a series of posts reviewing and explaining the Wasm specification. While the specification is well-written, it is not intended as an introduction to the format, and I haven't seen any good, comprehensive introductory materials that go over exactly what's in the spec. I think it would make more sense not to treat this as a graded deliverable, but if Dr. Ladd prefers it to be, that's fine too.

3 Preparation

While this is certainly an ambitious project, I think several of my experiences in and out of class have prepared me for it.

Relevant classes include Theory of Computation, Operating Systems, Assembly Language, Programming Languages, and the always-relevant Professional Practice. While I didn't submit completed versions of many steps of the Programming Languages scheme interpreter assignment, I did follow along with the project, and I feel I understand the techniques used.

I had previously intended to create a "fantasy console" for my capstone project, which would include a bytecode interpreter for a self-designed programming language. I eventually decided the scope of the project was too large, but in my attempts at prototyping, I completed a (buggy, unpleasant, incomplete) bytecode interpreter for a basic lisp.

I've made an emulator for the historical Chip 8 virtual console, which involved decoding a binary bytecode format.

I have submitted some minor contributions to Rust Analyzer, the Rust LSP. I wouldn't say I have a deep understanding of the code base, but since Rust Analyzer is essentially an incremental Rust compiler, that experience will be useful.

I have extensively reviewed the Wasm 3.0 specification and secondary sources describing the purpose and technical specifics of its included proposals. I believe I have the understanding of the format required to complete this project.

I have no useful prototype of the project, but I do have a repository, whose notes/ directory contains extensive sources on various technical aspects relating to the project.

4 Practice

I expect that this project will cover many of the topics traditionally covered in a Compilers class. Like a Compilers class, it will therefore tie together the theoretical, algorithmic, and concrete engineering aspects of computer science, and touch on material from nearly every other CS class (with the merciful exception of Networking). I will have the opportunity to review and expand upon the topics covered in those classes.

It will not interface with the operating system in the way that a normal compilers project would, but it will involve many of the low-level considerations brought up in the Operating Systems class.

Correct and understandable documentation of the project will require communication skills improved by several CS classes, particularly Software Engineering and Professional Practice. Preparation for the capstone presentation will similarly hone the in-person presentation skills we've covered.

5 Administration

The project will be completed in the Fall semester. I will register for 1 class, 4 credits.

A Grading rubric

I think the main axes of evaluation should be the quality of the compiler, the correctness of the compiler with respect to both Ruby and Wasm, the comprehensiveness of the implemented Ruby subset, quality of the documentation, quality of the demo, and quality of the presentation.

Rubric: 2.0 grade

Compiler quality

- Compiles a very small subset of Ruby programs, including basic statements, expressions and method declarations. Certain control structures, most data structures, and aspects of the object/class system are missing.
- Compiler successfully outputs Wasm programs for some correctly-formed programs
- Compiler crashes on many correctly-formed programs
- Compiler crashes without explanation on incorrect programs

Compiler correctness

- Many compiled Wasm binaries fail to run in targeted browsers
- Semantics of compiled programs differ significantly from expected Ruby semantics
- Compiler may only emit .wat files, requiring external processing to convert to Wasm binary format

Documentation quality

- Internal documentation is very incomplete, or is not inline with CS department coding guidelines
- Final compiler executable is undocumented or lightly documented. A user of the project would have to consult the project code to successfully run the project.
- No architectural explanation or justification of technical choices is included
- The available subset of Ruby is not documented or badly documented

Demo quality

- Produces a demo program that demonstrates successful execution, but may do nothing useful, or produce only a return value rather than any visible side effects
- Demo runs in some Wasm runtime, but may only run on desktop or with special setup
- Demo program is clearly buggy, frequently crashing or demonstrating logic errors

Presentation quality

- Presentation introduces the project, but doesn't describe it clearly, or leaves out important details
- Presenter demonstrates the functioning of the project in some way, but has significant technical challenges
- Presentation appears under-prepared. Presenter frequently pauses to consult notes
- Presenter is not able to respond effectively to audience questions

Rubric: 3.0 grade

Compiler quality

- Compiles a subset of Ruby programs, including most statements, expressions and method declarations. Most control structures, most data structures, and most of the object/class system are present.
- Compiler successfully outputs Wasm programs for most correctly-formed programs
- Compiler crashes on some correctly-formed programs
- Compiler gracefully terminates on incorrectly-formed programs, but does not produce many useful error messages

Compiler correctness

- Some compiled Wasm binaries fail to run in targeted browsers
- Semantics of compiled programs differ minorly or infrequently from expected Ruby semantics
- Compiler emits Wasm bytecode

Documentation quality

- Internal documentation is slightly incomplete, or differs somewhat CS department coding guidelines
- Final compiler executable is lightly documented. A user of the project might have to consult the project code for clarification on some
- Architectural explanation and justification of technical choices is included, but somewhat lacking
- The available subset of Ruby is documented, but may be unclear or missing some information

Demo quality

- Demo runs in browser and produces observable side-effects
- Interoperation with Javascript is lacking
- Demo program is mostly correct, rarely crashing or demonstrating logic errors

Presentation quality

- Presentation introduces the project, and describes it, but may assume too much prerequisite knowledge, or fail to be sufficiently detailed
- Presenter demonstrates the produced demo program, but may have minor technical problems
- Presentation appears reasonably well-prepared. Presenter may sometimes get stuck or consult notes
- Presenter is able to respond to many audience questions, but fails to answer some technical questions in a satisfying way

Rubric: 4.0 grade

Compiler quality

- Compiles a subset of Ruby that includes most or all control structures, many features of Ruby's object/class system, and many Ruby data structures
- Compiles almost all correctly-formed input programs
- Gracefully handles incorrect input programs
- Tracks source locations and provides basic error reporting in the lexer and parser

Compiler correctness

- Ruby programs using only the documented subset of the language compile and run identically in mainstream Ruby interpreters and my compiler, with the exception of documented semantic differences and minor edge cases
- Produced programs compile and run without error in browser Wasm implementations

Documentation quality

- Internal API documentation is complete and in line with CS department coding guidelines, perhaps with certain agreed-upon modifications
- Final compiler executable is documented from a user perspective, both in the project README and when running with the `-help` flag
- Documentation includes an architectural explanation, suitable for new project contributors with basic background knowledge. Architecture documentation includes explanation and justification of technical choices.
- The available subset of Ruby and semantic differences from mainstream Ruby interpreters are documented

Demo quality

- Demo runs correctly in targeted browsers, demonstrating intended functionality without logic errors or browser-reported exceptions
- Demo makes non-trivial use of Ruby data structures
- Demo demonstrates Wasm-Javascript interoperation

Presentation quality

- Presentation introduces project, rationale, and a technical/architectural overview
- Presentation demonstrates the operation of both the compiler executable and the demo project
- Presentation is delivered smoothly, and appears practiced
- Presenter responses to audience questions demonstrate understanding of the project and the underlying technology

B Tentative schedule

I don't think the date of the Board of Advisors meeting has been announced. I'm assuming November 15, which was the date of the Fall 2024 BoA meeting.

I have open availability for meetings, and will update this document when a meeting schedule has been decided.

I intend to produce a brief weekly report, including a list of backlog items completed that week.

Week 1: 8/31 - 9/6

- Install, compile and run reference programs – Hoot scheme compiler and a Ruby interpreter (probably mRuby, maybe CRuby, maybe both). Play with program output and begin to understand the project structure, entry points, etc.
- Collect auxiliary tools – Wasm bintools, desktop Wasm interpreter, etc.
- Create scaffolding project to run and debug compiled Wasm files in Chrome
- Walking skeleton – end-to-end lexing, parsing, code generation for a few constant values and operators I have most of this done already in my repo.
- Start to pin down value representation. Look to the reference programs for help.

Week 2: 9/7 - 9/13

- Finish the lexer or get close. Add some basic line tracking and error reporting. Rust analyzer might be good for this actually? Or the Crafting Interpreters book.
- Get some Ruby tool to produce an AST for input code. Copy or adapt that structure.
- Build out the parser. Take a look back at the Crafting Interpreters book for help. The main Ruby interpreters all seem to use Yacc and friends. Don't do that.

- Settle on representations for non-heap values. I think they're all going to be handled by slicing up the Wasm i31 type, like Hoot does. We're going to want immediate-value integers and big integers I think. Floats should be sort of free cause Wasm has them, and they all work mostly the same everywhere. Ruby documentation is a little vague on how it works, but the Ruby Spec Suite should be enlightening.
- Basic control structures – if / else, while. Wasm has some higher-level branching instructions, so this shouldn't be godawful.

Week 3: 9/14 - 9/20

- Local variable assignment and scoping. Pin down the Ruby scoping semantics. The Ruby Spec Suite should be useful.
- Global variable assignment. Wasm has facilities for both of these.
- Top-level method definitions. I have to figure out how Wasm modules need to map to Ruby definitions. I don't think it's going to be trivial. Ruby is also doing weird stuff with top-level definitions in general.
- Remind myself how closure implementations work. Take a peek at Hoot for sure. Crafting Interpreters also covers this, although I think they did something weird.
- Get a very clear understanding of Ruby's object system. There's some weird stuff here. Try to see how the Ruby interpreters implement it.
- Strings! I think I'm very willing to just use Javascript strings and ignore any associated problems.

Week 4: 9/21 - 9/27

- Wasm binary format. Ideally I'd get started earlier, but I can get pretty far generating .wat and converting separately I think. I know the actual wasm bytecode gets wrapped in object files, but I don't know anything about the structure. This will be needed for linking in-runtime facilities.
- Hash table implementation. This is the first real in-wasm runtime facility I'm creating. I think I want to write it in Rust/C and compile that to wasm as a separate project-compilation step, then link or paste in that wasm output, and call to it as a function. BUT that might produce obnoxiously-large binaries. I wonder what Hoot does here. Thank god I can stick everything in managed memory. The project would end here if I had to manage this myself.
- Start building out stdlib methods for core data structures as necessary. Ruby does some weird stuff for "primitives" so that they're still objects. I need to understand that by here.

Week 5: 9/28 - 10/4

- Hash tables part 2. I don't think I'll work out the issues with that in 1 week.
- Interned symbols
- Get started on class definitions and method dispatch. This is probably the thing I'm most vague on right now. It will definitely need hash tables though.

Week 6: 10/5 - 10/11

- Continue class/method definitions and dispatch. I assume this will be an ongoing war until the end of the project.
- Javascript FFI. I think this should be pretty easy, but Hoot added it a couple development versions in, so maybe it's not as easy as I think.
- Start the demo project. I want to start early so I can find any blocking missing features in my compiler. It wouldn't be a bad idea to write up a working comparison demo in Javascript or something.
- Review progress. Try and specify exactly what subset I want to implement by the end. Polish the architecture documentation based on the current state of the project.

Week 7: 10/12 - 10/18

- Bug fixes! I'm sure I'll have accumulated many by now.
- Build out stdlib with features needed for the demo.

Week 8: 10/19 - 10/25

- Bug fixes and remedial feature implementations.
- Start presentation planning and slides.

Week 9: 10/26 - 11/1

- Bug fixes and remedial feature implementations.
- Continue presentation planning and practice.
- Start capstone writeup.

Week 10: 10/2 - 10/8

- Bug fixes and remedial feature implementations.
- Polish the documentation.
- Present to Dr. Ladd. Incorporate feedback.
- Finish capstone writeup.

Week 11: 10/9 - 10/15

- Bug fixes.
- Final presentation practice.
- Presentation.

References

- [1] WC3 WebAssembly Working Group. *WebAssembly Core Specification 3.0. Draft*. May 15, 2025. URL (visited on 08/24/2025).
- [2] Andreas Rossberg et al. *GC Proposal for WebAssembly*. URL (visited on 08/24/2025).
- [3] WC3 WebAssembly Working Group. *WebAssembly Feature Status*. URL (visited on 08/24/2025).
- [4] V8 Project. *A new way to bring garbage collected programming languages efficiently to WebAssembly*. Nov. 1, 2023. URL (visited on 08/24/2025).
- [5] Sritely Institute. *Hoot: Scheme on WebAssembly*. May 23, 2023. URL.
- [6] Christine Lemmer-Webber. *Directly compiling Scheme to WebAssembly: lambdas, recursion, iteration!* May 23, 2023. URL (visited on 08/24/2025).
- [7] Ruby maintainers. *ruby.wasm*. URL (visited on 08/24/2025).
- [8] Artichoke maintainers. *Build the next Ruby for Wasm with Artichoke*. URL (visited on 08/24/2025).