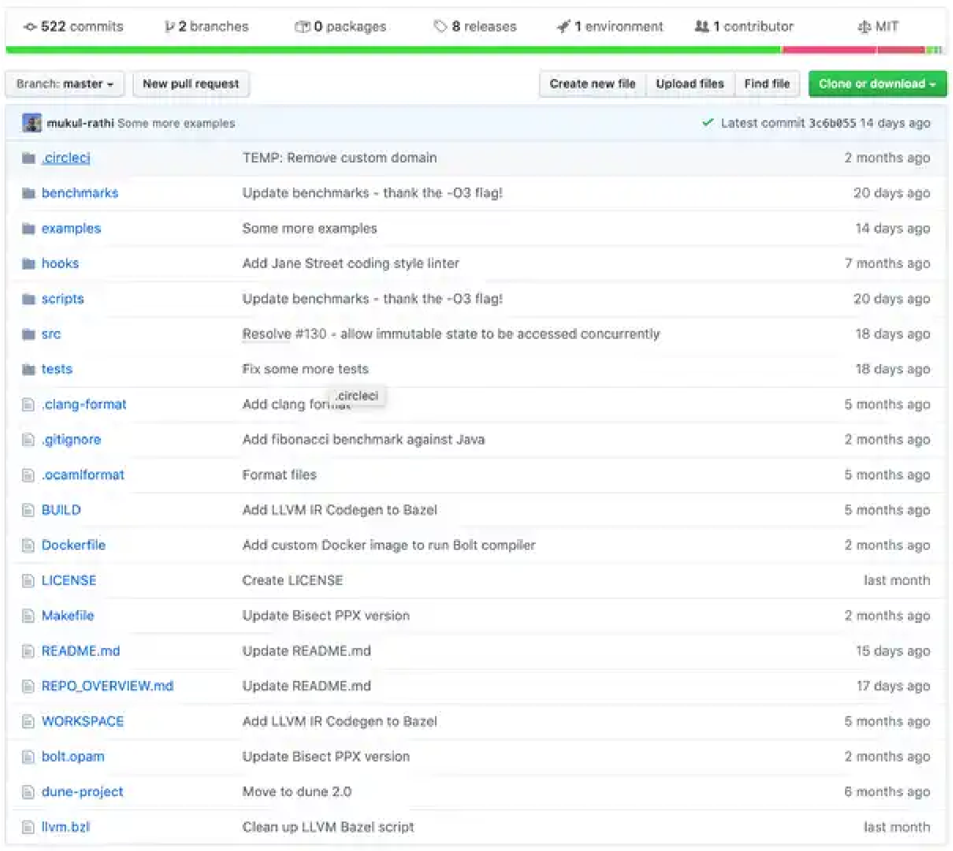
CREATING THE BOLT COMPILER: PART 2

So how do you structure a compiler project?

MAY 25, 2020 6 MIN READ

Last updated: January 10, 2021



522 commits | 2 branches | 0 packages | 8 releases | 1 environment | 1 contributor | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

mukul-rathi Some more examples — Latest commit 3c6b055 14 days ago

.circleci	TEMP: Remove custom domain	2 months ago
benchmarks	Update benchmarks - thank the -O3 flag!	20 days ago
examples	Some more examples	14 days ago
hooks	Add Jane Street coding style linter	7 months ago
scripts	Update benchmarks - thank the -O3 flag!	20 days ago
src	Resolve #130 - allow immutable state to be accessed concurrently	18 days ago
tests	Fix some more tests	18 days ago
.clang-format	Add clang format	5 months ago
.gitignore	Add fibonacci benchmark against Java	2 months ago
.ocamlformat	Format files	5 months ago
BUILD	Add LLVM IR Codegen to Bazel	5 months ago
Dockerfile	Add custom Docker image to run Bolt compiler	2 months ago
LICENSE	Create LICENSE	last month
Makefile	Update Bisect PPX version	2 months ago
README.md	Update README.md	15 days ago
REPO_OVERVIEW.md	Update README.md	17 days ago
WORKSPACE	Add LLVM IR Codegen to Bazel	5 months ago
bolt.opam	Update Bisect PPX version	2 months ago
dune-project	Move to dune 2.0	6 months ago
llvm.bzl	Clean up LLVM Bazel script	last month

TABLE OF CONTENTS

SERIES: CREATING THE BOLT COMPILER

Writing a compiler is like any other software engineering project in that it involves a lot of key design decisions: what language do you use, how do you organise your files in the repo, which tools should you be using? Most compiler tutorials focus on a toy example and choose to ignore these practical concerns.

With *Bolt*, I'd like to highlight a larger compiler and the design decisions I've made. If you're reading this and you work on an industrial compiler for a more mature language, please reach out on Twitter! I'd love to hear about the design decisions you took!

Use the right language for the job, not just the language you know best

"Write a compiler using language Y" (insert your favourite language) tutorials are a dime a dozen. It might seem easier initially to write a compiler using a language you know, as it's one less thing to learn, but this is only a short-term gain. Choosing the correct language is like learning to touch-type: sure it will be slower to start with, but just think of how much faster you'll be once you've got to grips with it!

JavaScript is a great language for web apps and easy to pick up for beginners. But would I write a compiler in it? **Frankly, no.** I'm not hating on JavaScript (I use it in this very site), it just doesn't suit our goal.

What do we care about for compilers?

- **Coverage** - we need to consider all possible Bolt expressions and make sure we handle all cases - it's no good if our compiler crashes on Bolt programs we forgot to consider. Does our language help us keep track of this?
- **Data representation** - how do we represent and manipulate Bolt expressions in the compiler?
- **Tooling** - does our language have libraries we can use for our compiler? There's a balance between learning by doing and unnecessarily reinventing the wheel.
- **Speed** - there are two different aspects. Firstly, how fast is the compiled Bolt code? Secondly, how fast is the compiler (how long does it take to compile the Bolt code)? There's a tradeoff - to get faster compiled code, you need to include more optimisation steps in your compiler, making the compiler slower.

There's no silver bullet: each compiler design inherently has its tradeoffs. I chose to primarily write my compiler in OCaml.

Why OCaml?

OCaml is a functional programming language with a powerful type system. You probably have two questions: why functional programming, and what do I mean by powerful type system?

In a large compiler, there's a lot of moving parts and keeping track of state makes our lives harder. Functional programming is easier to reason about: if you pass a function the same input it will always return the same output. With functional programming we don't have to worry about **side-effects** or state, and it lets us focus on the high-level design.

Another alternative is to write the compiler in Rust for performance reasons. Whilst you might have a faster compiler, I don't think the speed justifies the additional low-level details like managing memory that Rust requires you to track. Personally, since I'm not writing thousands of lines of Bolt code, I'm not too concerned with how long the Bolt compiler takes to compile programs.

Types let you pair program with the compiler

If you're coming from a dynamically typed language like JS or Python, OCaml's rich types can feel alien and may feel cumbersome. The way I think about types in OCaml is that they give the OCaml compiler more information about your program - the more you tell it, the more it can help you!

Coming back to our program of coverage, what we want to say is that a Bolt expression is *either* an integer, an if-else expression, a method call, a while loop etc. Normally to represent something like this, you would use an `enum` and a `switch` statement. In OCaml we bake this "enum" into our type system using *variant types*. We can encode the structure of each expression within the type! For example, to access a variable you only need to know its name `x`. To access an object's field you need to know both its name `x` and the field you're accessing `f`. We can then *pattern-match* based on each case: think of this like a `switch` statement on steroids!

Copy

```
type identifier =
| Variable of Var_name.t
| ObjField of Var_name.t * Field_name.t

let do_something id = match id with
| Var(x) -> ...
| ObjField(x,f) -> ...
```

I haven't even mentioned the best part. Because we've encoded our Bolt expression structures in a type, the OCaml compiler will check if we've covered all cases! That's one less thing for us to track!

Mukul Rathi
@mukulrathi_ · **Follow**

Adding support for generics to Bolt, which means modifying a core type definition in the compiler. #OCaml proceeds to flag up "pattern-not-exhaustive" warnings across 82 files - there's no way I'd catch every one without the type system!

9:00 AM · Apr 25, 2020

10 **Reply** **Share**

Read more on X

So OCaml takes care of coverage, and we've decided that we'll encode Bolt expressions as variant types, so that's the data representation sorted. OCaml also has great tooling for the lexer and parser stages of the compiler (discussed in the next post) which ticks off another of our criteria.

Targeting Performance with LLVM

We touched upon the fact that we don't really care about the performance of the compiler itself. However, we do want our compiled Bolt code to be fast (*it's in the name!*). As touched on in the previous post, we don't have to reinvent the wheel. By targeting *LLVM IR*, we can hook into the C/C++ toolchain and then get our optimisations for free!

LLVM provide APIs for language authors to generate LLVM IR - the *native* API is in C++. LLVM also offers bindings in other languages - they are identical, just replace the C++ syntax with whichever language you're using. LLVM actually offers OCaml bindings.

Why is the compiler backend written in C++?

A natural question you might ask is: well why didn't you write everything in OCaml and use the LLVM OCaml bindings?

LLVM's OCaml bindings only map some of the C++ API. At the time of implementation there was no support for implementing memory fences (a machine instruction needed to implement locks correctly) so I was forced to write this part of the compiler in C++. I was also experimenting with some other fancier memory consistency instructions only present in the native C++ API.

I want to be frank with you, the OCaml LLVM bindings will likely be sufficient for your language and I **encourage you to use that instead**. See, the tradeoff with the approach I took is that now we have to pass data between the OCaml compiler *frontend* and the C++ compiler *backend* using Protobuf. Yes the C++ API has more power, but it results in a more complex compiler.

Like I said, the LLVM API is the same in OCaml and C++ (apart from syntax), so this tutorial still applies, just skip the Protobuf post and use the llvm OCaml package!

Software Engineering Methodology

The repository contains more information about how the compiler is structured in REPO_OVERVIEW.md. Most of these are general software engineering tips so I'll keep this brief. (If you want more detail, feel free to reach out!)

Firstly, the repository is structured to be *modular*. Each stage of the compiler has its own library, whose documentation you can view. Functions are grouped into *modules* (the `.ml` file provides the implementation for the module, and the `.mli` file provides the module interface). You can think of each module as performing a certain role within a stage e.g. `type_expr.ml` type-checks expressions, `pprint_parser_tokens.ml` pretty-prints parser tokens. It makes each module more focused, and avoids monolithic hundreds-of-lines-long files that are hard to read.

To build these files I use the Dune build system for OCaml (I have a blog post explaining it) and the Bazel build system for C++. For large repositories, manually compiling each file (e.g. by running `clang++ foo.cpp`) and linking files that depend on each other is nigh on impossible - build systems automate this for us (running one `make build` command will compile all the files in the repository). One of the main benefits of Bazel is that the dependencies are all self-contained so will work across machines.

For testing, the main library I use is the Jane Street's **Expect tests** library. This library is *really easy* to write tests for as it autogenerates the expected output. I have a blog post on testing in OCaml that covers this. The post also explains how I set up Continuous Integration (where the tests are run and documentation is generated on each commit to the repository).

Mukul Rathi
@mukulrathi_ · **Follow**

PPX_Expect by
@JaneStreetGroup
is a great testing library for #OCaml, I'm using expect tests for my pretty-printed typed ASTs and I love how "dune runtest --auto-promote" fills out the expected result for me!

Watch on X

I also automate common tasks using a `Makefile` and some scripts. One tool I would highly recommend is an autoformatter - I use `ocamlformat` for the OCaml code and `clang-format` for the C++ code - this formats your code for you so you have pretty looking code for free! You can automate it with the git pre-commit hook in the repo (which will lint and format your code every time you're about to commit) or via IDE format-on-save integration. One final tip: use VSCode's OCaml IDE extension or the equivalent for your IDE - whenever you hover over a function it will display its type signature and any documentation comments associated with it.

Summary

In these first couple of posts, I've explained where Bolt sits in the spectrum of programming languages, and the compiler design and software engineering decisions made. In the next post, we'll actually get to building this. Before we do, I have a couple of action items for you:

- Get up to speed with OCaml. Real World OCaml is a great free resource.
- Fork the repo. Have a quick high-level scan but don't worry about the details just yet! We'll break down each stage of the compiler in its own post, and dedicate entire posts to the more complex language features.

Share This On Twitter

If you liked this post, please consider sharing it with your network. If you have any questions, tweet away and I'll answer :) I also tweet when new posts drop!

PS: I also share helpful tips and links as I'm learning - so you get them **well before** they make their way into a post!

Post Follow @mukulrathi_

SERIES: CREATING THE BOLT COMPILER

Part 11: Adding Inheritance and Method Overriding to Our Language

← How I wrote my own "proper" programming language

Writing a Lexer and Parser using OCamllex and Menhir →