Crash Course in julia

Introduction

- What this is
  - Get you set up to write and share Julia code without doing mistakes that bite you down the line.
  - Understand the design of Julia and its use case
- What this is not
  - An introduction to the Julia programming language itself.
  - https://docs.julialang.org/en/v1/
  - https://julialang.org/learning/

Overview

1. Getting started
   - Install Julia
   - Install vscode IDE
   - Set up the flight deck
2. What is Julia? What is the difference between Julia, C/C++, and Matlab?
3. Starting a new project
   - Basic structure and coding conventions and Github integration
   - Glimpse at package release
4. Developing in Julia
   - Workflow
   - How to debug (spoiler: without a debugger)
5. Writing Julia
   - Multiple dispatch, types, composability, and functional programming
   - Macros
   - Broadcast and GPU code

Getting Started

Install Julia

1. Go to the webpage, download for your architecture
   - https://julialang.org/downloads/
   - LTS vs latest Julia: Always use latest! Always update! LTS is for corporate development
   - Add Julia to PATH in .bashrc, done.
   - Julia is self-contained and has no system dependencies besides libc.
2. Juliaup
   - https://github.com/JuliaLang/juliaup
   - Linux + Mac: `curl -fsSL https://install.julialang.org | sh`
   - Windows: `winget install julia -s msstore`
   - Windows WSL users: You only need to have Julia installed in WSL
   - Juliaup is upstream in the Linux distros

Install Julia

- Compile Julia 🐉
  - `git clone git@github.com:JuliaLang/julia.git`
  - `make -j16 VERBOSE=1 USE_BINARYBUILDER=1 binary-dist`
  - Use prebuilt binaries for bootstrapping

Install vscode

- https://code.visualstudio.com/
- Essential extensions
  - WSL, Remote-SSH
  - Julia
- Optional extensions
  - Unicode Latex
  - Github Copilot
  - Grammarly
  - Live Share to do peer coding/debugging
  - GitLens
- Remember Super Key Combo: CTRL + SHIFT + P

What is Julia?

What is Julia?

- New: 10 years (C 50 years, C++ 37 years, Fortran 65 years, Python 31 years, Matlab 40 years)
- Interactive (like Python)
- Fast compiled execution (like C/C++)
- Focus on science (like MATLAB/Fortran)
- Solves the two-language problem
  - Prototyping and production
  - High-level and low-level
- Garbage collected memory allocation (no segfaults)
- It’s free. It’s open source. The source code is required to run it.
- It’s portable: Single code to run on various CPUs (x86, Power9, ARM) and GPUs (Nvidia, AMD, Intel)
- Why was this not done before?

What is Julia?

- First Turing complete computers
  - Turing completeness: A computer that can execute any algorithm iff. it is a universal Touring machine.
- How to actually program them?
  - In science: map mathematical models to machine instructions by humans !
- Tedious to do manually
- Can’t we generate machine instructions?
  - Slow in generation (Compile time)
  - Slow generated code (Runtime)
  - Programming languages

What is Julia?

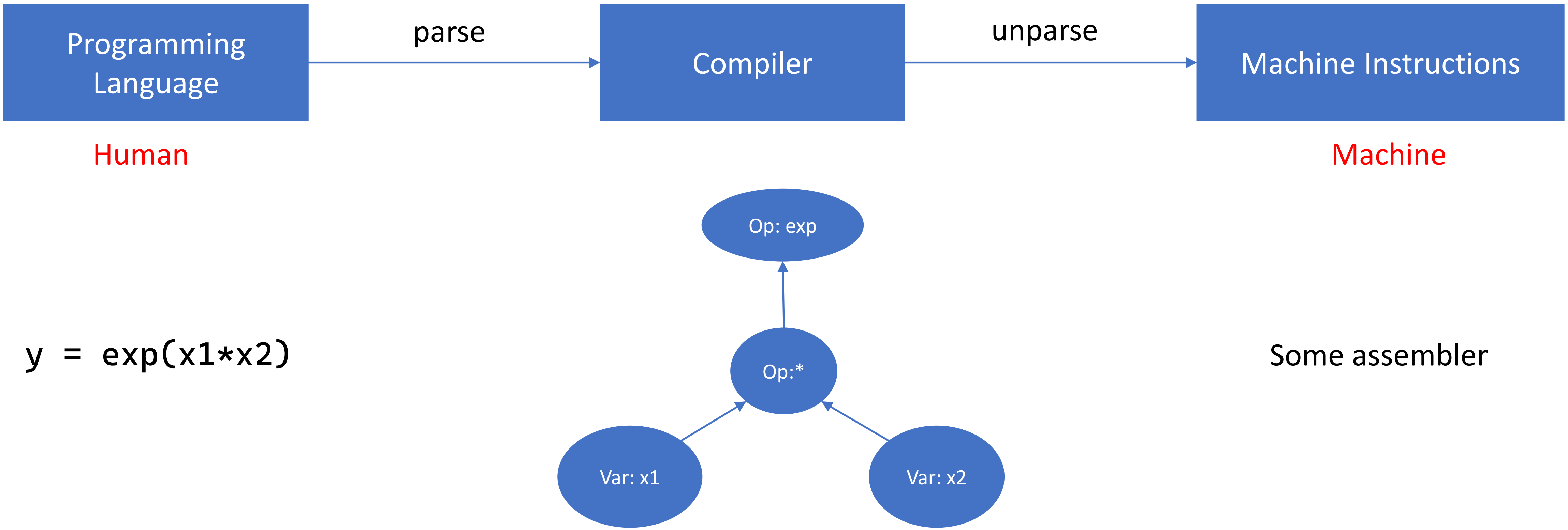
Compilers and Languages

- Human languages have meaning through structure. What is that structure?
  - Chomsky grammars, universal grammar (Noam Chomsky, "Three models for the description of language," in IRE Transactions on Information Theory, vol. 2, no. 3, pp. 113-124, September 1956)
- Compilers: Language translation between humans and computer hardware
  - Regular expressions, context-free grammars
  - Parsing of text is critical for speed and machine code language design
  - Languages should be expressive and compact, yet easy to parse
- Algorithms that transform semantics or expressions
  - Modeling, parallel paradigms, automatic differentiation, profiling
  - Code transformations, runtime injection through linking

What is Julia?

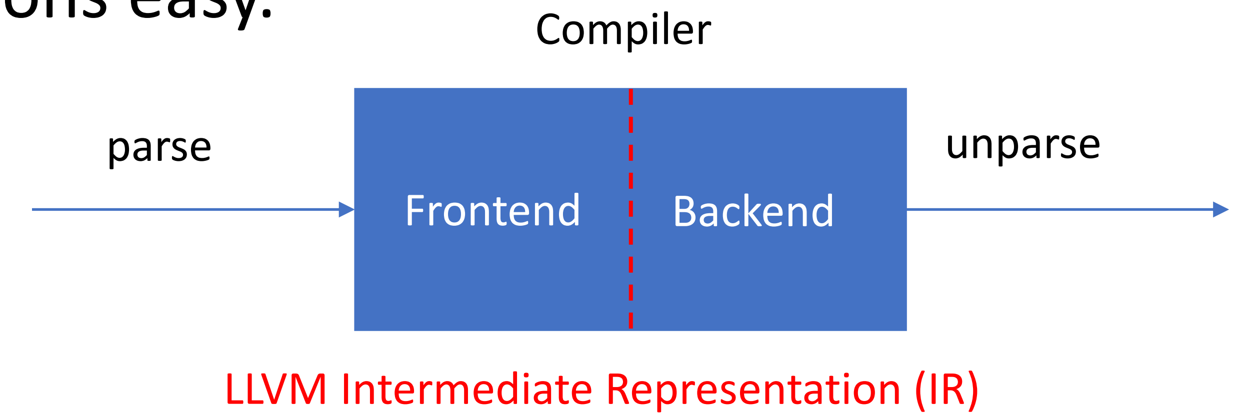
Design of Compilers

- At the core of language models are abstract syntax trees. Simplified example with a directed acyclic graph (DAG) of a program execution.

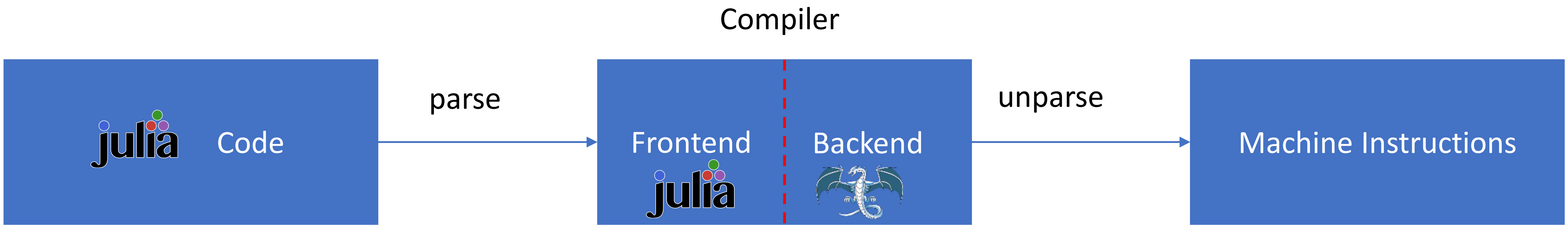


LLVM

- From Wikipedia: “LLVM is a set of compiler and toolchain technologies.”
- Started in 2003 at Urbana Champaign, Apple hired lead developer in 2005
- LLVM implements a backend of a compiler design
- First popular compiler that standardized LLVM. Goal: Make frontend implementations easy.



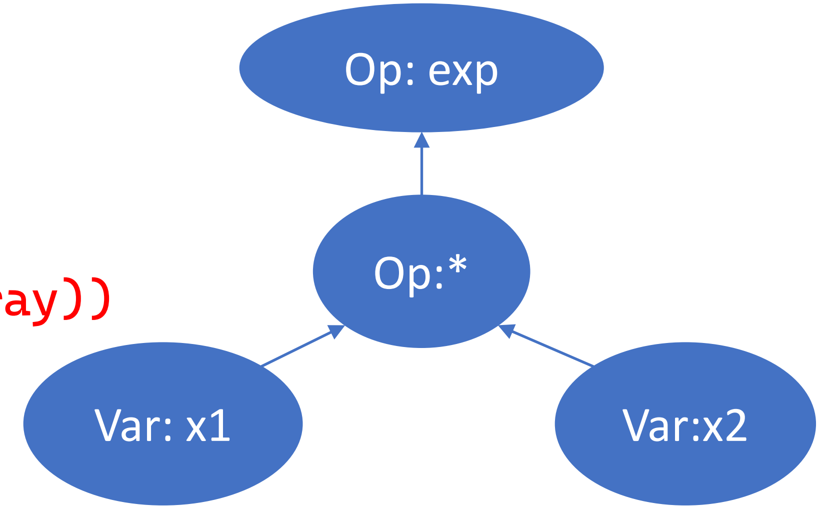
LLVM IR



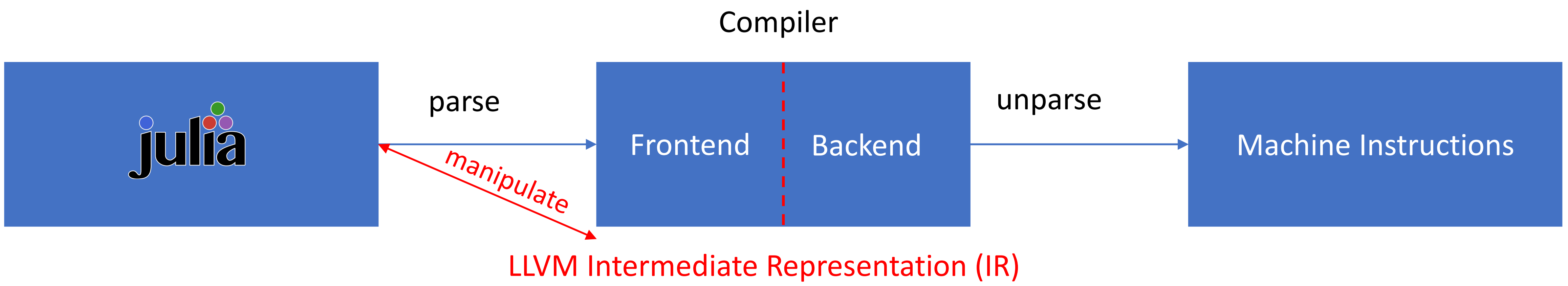

```
using CUDA
using ForwardDiff
function f(x)
  y = exp(x .* x)
end
x = [2.0,2.0]
@code_llvm(f(x))
@code_llvm(f(x |> CuArray))
@code_llvm(jacobian(f, x |> CuArray))
```

LLVM Intermediate Representation (IR)

x86, arm, CUDA, ROCm, Cerebras



Julia



- Julia leverages LLVM to implement a just-in-time compiled language with native metaprogramming and code reflection. Code is compiled at runtime.
- Language support for IR and expression tree manipulation
- Advantage: Highly flexible (JIT) C/C++-like performance (LLVM backend)
  - Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift
- Disadvantage: LLVM was not created with JIT in mind
- See interpreted languages: Trying to reduce compilation (Python)

What is Julia?

- Julia is a JIT-compiled language leveraging LLVM and its IR
- Compile time and runtime become one
- Trade-off between compile time and runtime?
- WIP: Julia stores object (machine) code in Julia 1.9 (in alpha)
- Now: Julia has a precompilation stage
- Julia cannot precompile what is unknown at compile time!!!
- Main causes of slow compilation
  - Type stability (e.g., unknown return type of functions)
  - Missing type annotations
  - Symptom: No clear separation between setup and execution, bonus: great for GPUs
- Tools: Profile.jl, BenchmarkTools.jl, PProf.jl, Cthulhu.jl

Starting a New Project

Starting a New Project

- Environments
- Package file structure
  - `Project.toml`
  - `Manifest.toml`
  - `src` source code folder
  - `test` continuous integration folder
  - `doc` documentation folder
- Code structure
- GitHub Actions to run tests on GitHub servers

Starting a New Project

Environments

- Start with `Pkg.generate("MyProject.jl")`
- Project names: What does the package DO, CamelCase
- Defined in `Project.toml` (Dependencies)
- Instantiated in `Manifest.toml` (Reproducibility)
- Reproducibilty and sharing/composability
- Global environment is always active (later), should be as empty as possible ❗

Starting a New Project

- Demo, run the Julia code from before
- CUDA.jl is optional
- 🐉 Mac users Metal.jl https://github.com/JuliaGPU/Metal.jl
- 🐉 Intel Xe integrated graphics: oneAPI.jl: https://github.com/JuliaGPU/oneAPI.jl
- No need to install CUDA manually!

```
using CUDA
using ForwardDiff
function f(x)
  y = exp(x .* x)
end
x = [2.0,2.0]
@code_llvm(f(x))
@code_llvm(f(x |> CuArray))
@code_llvm(jacobian(f, x |> CuArray))
```

Starting a New Project

- Language features are implemented in Module `Base`
- Most important object type in Julia is `Array <: AbstractArray`
- Various other array types:
  - `CuArray <: AbstractArray`,
  - `Dual <: AbstractArray`,
  - `MyUQArray <: AbstractArray`
- `Vector{T} = Array{T,1}, Matrix{T} = Array{T,2}`
- Type parameters `{}`
- Broadcast operator `.`

Crash Course in julia

Part 2

Summary Part 1

- Getting started
    - Installation of Julia
    - VSCode + extensions
- What is Julia
    - Compiler + Runtime
    - LLVM and expression transformations
- Starting a new project
    - Folder layout
    - Package manager and `Pkg.generate(“Neighborhoods.jl”)`
    - Example code with multiple dispatch on `AbstractArray` types.

Overview

1. What is Julia? What is the difference between Julia, C/C++, and Matlab?
2. Getting started
   - Install Julia
   - Install vscode IDE
   - Set up the flight deck
3. Starting a new project
   - Basic structure and coding conventions and Github integration
   - Glimpse at package release
4. Developing in Julia
   - Workflow
   - How to debug (spoiler: without a debugger)
5. Writing Julia
   - Multiple dispatch, types, composability, and functional programming
   - Macros
   - Broadcast and GPU code

Developing in Julia

Developing in Julia

- Writing C/C++
  - Write code, recompile, execute, repeat
- Writing in Julia
  - Write code, ?, execute, repeat
  - Restarting Julia is slow with `julia --project mycode.jl` as the entire code has to be recompiled
  - Advantage: only code that is executed is compiled
  - Disadvantage: code is recompiled (including dependencies) at every run, testing is hard, and unexpected behavior when composing Packages
  - Option to create binaries with `PackageCompiler.jl` 🐉
    - Required in competitions like ARPA-E
    - Not needed for benchmarks

Developing in Julia

- `Profile`
  - Included in Julia Base
  - `@time`, measure wallclock time
  - `Profile.Allocs.@time`, trace allocations (since Julia 1.8)
- `BenchmarkTools.jl`
  - `@btime`, stochastic profiling for small snippets
- `PProf.jl`
  - Call graph and large profiling with flame graph

Developing in Julia

Reduce compile time

1. Emacs, vim, REPL: Revise.jl, install in the global environment
2. vscode, REPL: Julia -> Settings -> Execution

Demo

- Implement a neighborhood package, which evaluates $f(x)$, $f(x-\epsilon)$, and $f(x+\epsilon)$ in module, ϵ being machine precision
- Observe recompilation

Developing in Julia

- Add `neighborhood(f,x)` function
- Implement differentiation of a neighborhood
- Demo
  - Distinguish between functions and methods
    - Functions are a collection of methods with the same name (see `methods()`)
    - Functions describe the **what action** and arguments are for **objects** and **how actions are applied**
    - Observe dynamic dispatch, enabling composition

Developing in Julia

- Add `neighborhood(f,x)` of function f at point x
- Demo
  - Distinguish between functions and methods
    - Functions are a collection of methods with the same name
    - Functions describe the **what action** and arguments are for **objects** and **how actions are applied**
    - Observe dynamic dispatch enabling composition
- Summary
  - Used dynamic dispatch to implement derivatives of neighborhoods
  - Extended ForwardDiff with neighborhoods

Developing in Julia

- Make use of composability
- Write only pure functions!
  - No side effects
  - No global variables
- Can lead to a mess!
- Think about who uses what functionality and where
- Think about readability more than fancy composition
- When to restart Julia is tricky, but in general
  - Functions can be redefined; types cannot.

Developing in Julia

- Included minimal CI
- Write tests!
- Demo

Developing in Julia

Profiling

- Use Profile
- @time: Timing
- @btime: Statistical timing without compile time on small code snippets
- Available for memory in Julia 1.8 Profile.Allocs

Share Code

Share Code

- Julia is tightly integrated with GitHub
- User code should include Project.toml, src, test, and optionally Manifest.toml
- Git clone, run with Julia –project
- Include a license file
- Include a README.md

Share Code

- What is missing?
  - Documentation: Look into Documenter.jl
    https://documenter.juliadocs.org/stable/
  - Release your package to the public Julia repository
    https://github.com/juliaregistries/registrator.jl
- Everything is integrated into GitHub

Summary

- What is special about Julia’s design? Frontend for LLVM IR JIT
- Compile time vs. runtime
- vscode setup, remotes, REPL execution
- GPU code using broadcast operator or writing kernels
- Start a new project
- Functions, functions, functions
- Objects struct are only used for context, state, and preallocation
- Macros `@macro(expr)` 🐉🐉🐉
- Artifacts defined in `Artifacts.toml`
- Documentation with `Documenter.jl`

Summary

- Python both ways
  - https://github.com/juliapy/pyjulia
  - https://github.com/JuliaPy/PyCall.jl
- C interface also both ways
  - Calling C in `Base`
  - Julia has a C API 🐉
- C++ interface is difficult 🐉
- Macros `@macro(expr)` 🐉🐉🐉
  - Increases compile time ❗
- Artifacts defined in `Artifacts.toml`
  - Host artifacts on the web (e.g., binaries, data)
- Documentation with `Documenter.jl`
- Plotting with `Plots.jl`, benchmark time to first plot (TFP)
  - Interface to multiple plotting libraries including `matplotlib`
- Julia is still a young language, with the ecosystem still in development
  - Manage your expectations

Why invest in Julia (nor not)

- Julia is the right tool for the right problem (scientific computing)
- Even if Julia won’t make it, something similar will come
- Reduce development cost in a business (science), where software is more transient than in other industries
- No big company behind it. In comparison with Python, Julia has a minuscule community
- Alternatives: NUMBA, Rust, Python