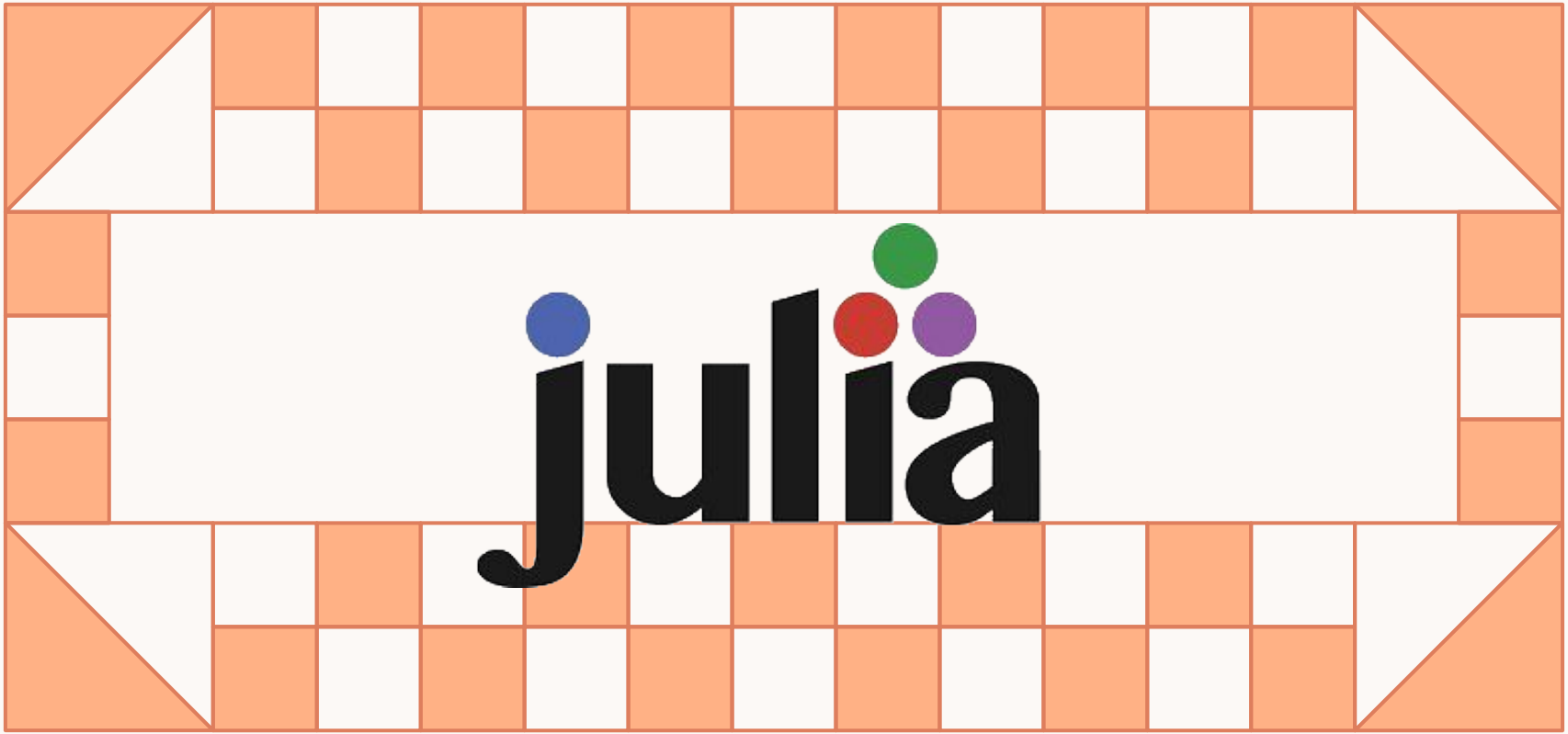
Cuyler, Deven, and Sam



BACKGROUND

- Started development in 2009
- Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman at MIT
- First publicly announced and released as open source in February 2012
- Aim was to address users experience between code readability and execution speed

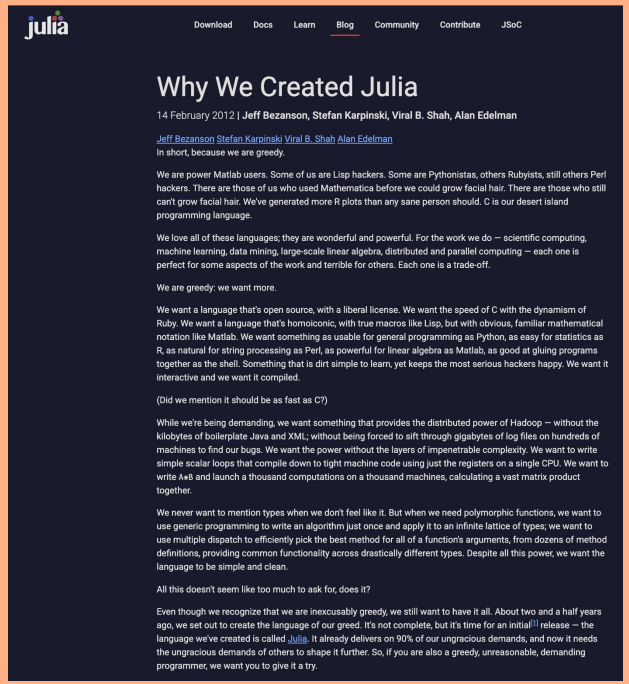
- Basically, take all the best parts of Python, Ruby, Matlab, Perl, Lisp, R, and form into one language
- With the speed of C and all compiled
- Options, options, options

“In short, because we are greedy.”

Goals

Create a language that combines all the best features of others like:

C: Speed and performance
Ruby: Dynamism
Lisp: For being homoiconic with true macros
Matlab: Mathematical notation and power in linear algebra
Python: Usability in general programming
R: Ease of use in statistics
Perl: Natural string processing capabilities
Hadoop: Distributed computing power



julia

Download Docs Learn Blog Community Contribute JSoC

Why We Created Julia

14 February 2012 | **Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman**

Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman
In short, because we are greedy.

We are power Matlab users. Some of us are Lisp hackers. Some are Pythonistas, others Rubyists, still others Perl hackers. There are those of us who used Mathematica before we could grow facial hair. There are those who still can't grow facial hair. We've generated more R plots than any sane person should. C is our desert island programming language.

We love all of these languages; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

We are greedy: we want more.

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

While we're being demanding, we want something that provides the distributed power of Hadoop — without the kilobytes of boilerplate Java and XML; without being forced to sift through gigabytes of log files on hundreds of machines to find our bugs. We want the power without the layers of impenetrable complexity. We want to write simple scalar loops that compile down to tight machine code using just the registers on a single CPU. We want to write A*B and launch a thousand computations on a thousand machines, calculating a vast matrix product together.

We never want to mention types when we don't feel like it. But when we need polymorphic functions, we want to use generic programming to write an algorithm just once and apply it to an infinite lattice of types; we want to use multiple dispatch to efficiently pick the best method for all of a function's arguments, from dozens of method definitions, providing common functionality across drastically different types. Despite all this power, we want the language to be simple and clean.

All this doesn't seem like too much to ask for, does it?

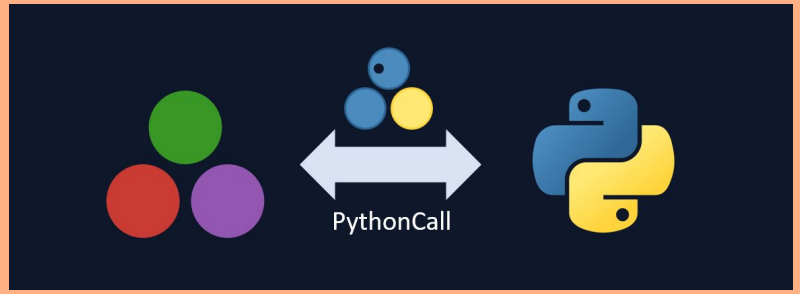
Even though we recognize that we are inexcusably greedy, we still want to have it all. About two and a half years ago, we set out to create the language of our greed. It's not complete, but it's time for an initial[1] release — the language we've created is called Julia. It already delivers on 90% of our ungracious demands, and now it needs the ungracious demands of others to shape it further. So, if you are also a greedy, unreasonable, demanding programmer, we want you to give it a try.

Goals

Solve the “Two-Language Problem” (Speed +Readability)

Historically, prototyped the algorithms in high-level, but slow languages and then had to rewrite them in low-level, fast languages

They want a language where prototyping is easy and execution is fast!



Features

Support for Abstraction

- Instead of object oriented classes containing methods, Julia uses multiple dispatch
- Chooses best/fastest method based on parameters
- Optimized performance, efficient

```
julia> struct Dog end; struct Cat end

julia> meet(a::Dog, b::Dog) = "The dogs play together"
       meet(a::Dog, b::Cat) = "The dog chases the cat"
       meet(a::Cat, b::Dog) = "The cat hisses at the dog"
       meet(a::Cat, b::Cat) = "The cats ignore each other"

julia> meet(Dog(), Cat())
"The dog chases the cat"
```

Features

Just-In-Time Compilation

- Convert high-level → machine code during runtime
  - Code is compiled when run
  - Ties with multiple dispatch
- Compiles functions at first execution
  - Based on parameters passed
- LLVM (Low Level Virtual Machine)
  - Compiles code from user
- Ties with REPL tab
  - Allows users to input lines of code at a time
- GOAL: Fast performance (comparable to C)

```
julia> f(x::String, y::String) = println(x, y)
f (generic function with 2 methods)

julia> f(x::Int, y::Int) = x + y
f (generic function with 3 methods)

julia> f(2, 3)
5

julia> f("Hello", "World")
HelloWorld
```

Features

Data Types

- Primitive and non-primitive
- Dynamic Typing
  - Variables do not need type declarations
  - Can assign type to improve optimization

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of type Int64

julia> (1+2)::Int
3
```

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

Data Structures

- Built-in structures
  - Ex. Arrays, dictionaries, sets, etc.
- DataStructures.jl
  - Official Julia package
  - Contains advanced data structures

```
julia> Float32[1, 2.3, 4//5]
3-element Vector{Float32}:
 1.0
 2.3
 0.8
```

Features

- **Numerical/Scientific Computing**
  - Math-friendly syntax
  - Rich sets of numerical libraries
  - Type Inference
  - Efficient Memory Management
  - Interoperability
  - Distrubuted Computing / Parallelism

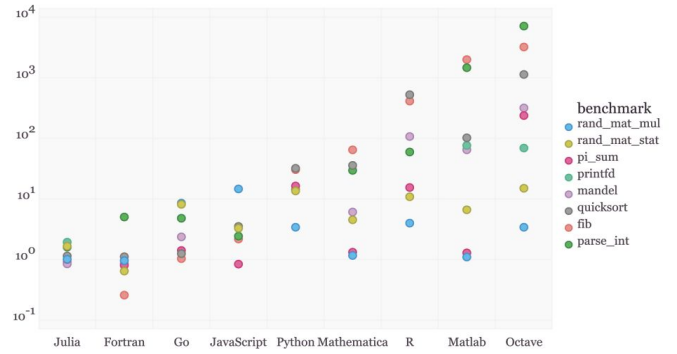
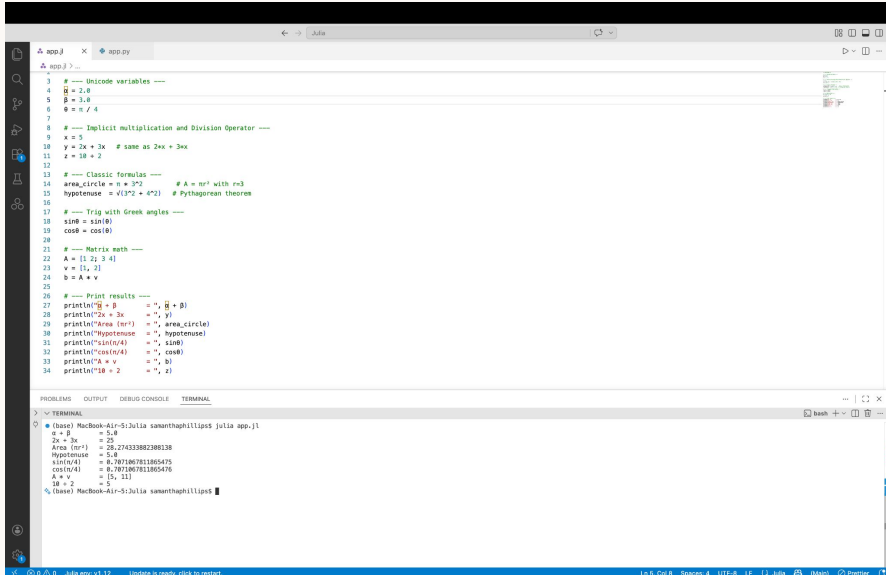
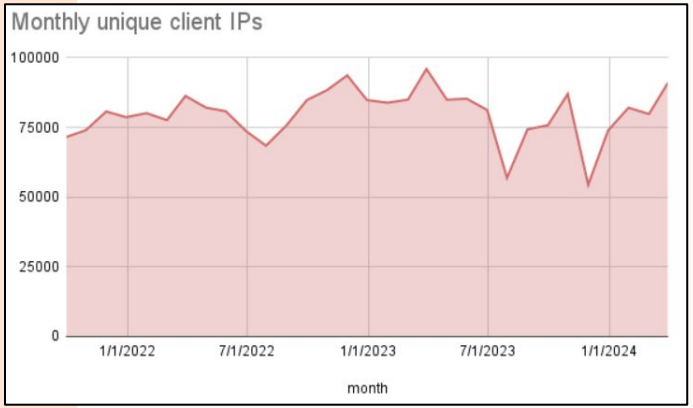


Fig. 5 *Performance comparison of various languages performing simple microbenchmarks. Benchmark execution time relative to C. (Smaller is better; C performance = 1.0.)*



Users

- **Over 100 million downloads**
  - Community all over the world
  - Over 12,000 Julia packages
- **In 2024, ~ 100,000 active users**
- **JuliaCon**
  - Annual conference that connects community
  - Technical talks, workshops, networking
- **Researchers, data scientists, and engineers**



Industry Users

- **Prioris Ai**
  - “We focus on building bespoke, interpretable, predictive models for the life science industry”
- **NASA**
  - For modeling spacecraft separation dynamics (*15,000 times faster than before with Simulink/MATLAB and the Brazilian INPE for space mission planning and satellite simulation*).
- **Climate Modeling Alliance**
  - A climate model built in Julia and designed from the outset to leverage GPU acceleration and modern software engineering practices to overcome the limitations of traditional climate models.

DEMO Slide

MORE OVERVIEW OF LANGUAGE

https://cheatsheet.juliadocs.org/

REFERENCES

1. https://www.stochasticlifestyle.com/like-julia-scales-productive-insights-julia-developer/
2. https://enccs.github.io/julia-for-hpc/
3. https://www.geeksforgeeks.org/julia/julia-concept-of-parallelism/
4. https://julialang.org/
5. https://discourse.julialang.org/t/some-julia-growth-usage-stats/112547
6. https://www.pass4sure.com/blog/top-10-real-world-applications-of-julia-programming/
7. https://prioris.ai/
8. https://discourse.julialang.org/t/julia-notable-uses/86130
9. https://julialang.org/jsoc/gsoc/clima/
10. https://machaddr.substack.com/p/julia-programming-language-a-powerful
11. https://julialang.org/assets/research/julia-fresh-approach-BEKS.pdf
12. https://julialang.org/blog/2012/02/why-we-created-julia/