Julia: a strange place in language space

- "bare metal" (reasoning about machine)
- abstract (reasoning about math)
- statically analyze program "early"
- "late" compilation and error-catching

Languages:
- Scala,
- Java,
- C++
- Haskell
- C#
- Fortran
- Python, R, etc.
- Julia
- Lisp
- C
My wish-list from last time:

- define a subset of a well-known syntax, such as Python
- that JIT-compiles to CPU, GPU, and maybe FPGA
- that can be freely mixed with surrounding code
- that is immutable, maybe total-functional

- with special handling of Python’s imperative statements and restrictive lambdas

- using interval arithmetic as data types

- separate mathematics from performance annotations in CSS
Julia is almost everything I wanted:

My wish-list from last time:

- define a subset of a well-known syntax, such as Python or Julia.
- that JIT-compiles to CPU, GPU, and maybe FPGA.
- that can be freely mixed with surrounding code: check.
- that is immutable, maybe total-functional: check for Julia functions ending in “!”
- with special handling of Python’s imperative statements and restrictive lambdas: Julia syntax is expression-only and easily defines complex lambdas.
- using interval arithmetic as data types: Julia’s metaprogramming supports this kind of inspection.
- separate mathematics from performance annotations in CSS: Julia doesn’t do this, but its metaprogramming would help build a demonstration project.
However...
### Potential stumbling blocks for HEP analysts

<table>
<thead>
<tr>
<th>Julia behavior</th>
<th>HEP expectation</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays range from 1 to $N$</td>
<td>0 to $N - 1$ from C arrays, C++ vectors, and Python lists</td>
</tr>
<tr>
<td>histogram (built-in <code>hist()</code>) is exclusive on low edge and inclusive on high edge</td>
<td>inclusive on low edge and exclusive on high edge (HBOOK, PAW, ROOT, Numpy)</td>
</tr>
<tr>
<td>multidimensional arrays are column-major</td>
<td>row-major C/C++, Numpy default, GPU</td>
</tr>
<tr>
<td>no OOP-like fluent syntax:</td>
<td>physicists have become accustomed to thinking in terms of objects</td>
</tr>
<tr>
<td><code>event.track(2).hit(12)</code> → hit(track(event, 2), 12)</td>
<td>C/C++ <code>else if</code>; Python <code>elif</code></td>
</tr>
<tr>
<td><code>elseif (no space)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
- C/C++ uses column-major order for multidimensional arrays.
- Python uses row-major order for multidimensional arrays.
- Numpy uses row-major order by default, but can be changed.
- HBOOK, PAW, ROOT use inclusive on low edge and exclusive on high edge for histograms.
- Other libraries like Numpy use inclusive on low edge and inclusive on high edge for histograms.
Potential issues for building large projects

- Since everything is JIT-compiled, some errors are discovered only when they’re encountered, such as `UndefVarError`. Unit tests need to fill in for missing “compiler checks,” and it’s hard to guarantee coverage of all potential uses (just like Python programming or C++ templates).

- **Function** is not a parameterized type, so functions passed as arguments can’t be constrained by method signature.

- Without methods in data structures (classes), the namespace fills up: I might think I’m defining a new function for my custom data structure, but I’m actually adding methods to a global function of the same name.

- Would need formal interfaces (contracts) to get an error if an implementation is incomplete.

- Do all packages need to be GitHub repositories ending in `.jl`?