A Tour of Julia

Erik Engheim
A Tour of Julia
The Goldie Locks language

Erik Engheim
@erikengheim
It SUCKS writing Fortran!

But it runs so SLOW!

Do Python! fun, flexible and quick to write
Yeah, got to be **STATICALLY TYPED** to run fast

Hahaha yeah right, as if that is possible!

what if there was a way of making **DYNAMIC** languages run fast?
Julia
Creators
Stefan Karpinski
Viral Shah
Jeff Bezanson
Professor Alan Edelman
ACT STATIC WHEN POSSIBLE
Static behavior 90% of the Time
What is Julia?
Fundamental attributes of the language

• General Purpose
• Dynamically Typed
• High Performance, JIT
• Multi-platform
• Numerical Language
Where is Julia used?
Celeste Project
Creating a catalog of celestial objects

- Lots of photos of the sky with no order
- Brightness, rotation of visible objects
- 9300 Intel Xeon Phi processors
  - 650,000 cores
  - 1.54 petaflops
- 178 terabytes
CliMA
Climate Modeling Alliance

- **New Earth Systems Model**
  - Higher resolution simulation
  - Machine learning

- **Scientists at Caltech, MIT, NASA JPL**

- **Open Source on GitHub**

- **Performance:** Few percent away from Fortran
Yes you can do other things than science in Julia!
simulate_launch(rocket, Δt; max_duration = 2000)
Returns a rocket object giving all state after all fuel is spent. You can specify a maximum duration `max_duration` of the flight in seconds. This is practical to avoid the simulated launch never terminating.

function simulate_launch(spaceship::SpaceVehicle, Δt::Number; max_duration::Number = 2000)
    t = 0  # start time
    ship = copy(spaceship)
    while ship.active_stage isa Rocket
        while propellant(ship) > 0 && t <= max_duration
            boosters = sideboosters(ship)
            if !isempty(boosters) && sum(propellant.(boosters)) <= 0
                detach_sideboosters!(ship)
            end
            update!(ship, t, Δt)
            t += Δt
        end
        stage_separate!(ship)
    end
    ship
end
<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language Tour</strong></td>
<td>Functions, variables, loops, if-statements, arrays</td>
</tr>
<tr>
<td><strong>Programming Language Trade-Offs</strong></td>
<td>Why are dynamic languages slow? Boxing, memory fragmentation</td>
</tr>
<tr>
<td><strong>What is the Secret?</strong></td>
<td>Just in time compilation? Language Design?</td>
</tr>
<tr>
<td><strong>JIT Code Generation</strong></td>
<td>Vector dot product, lowering, abstract syntax tree, LLVM bitcode, native assembly</td>
</tr>
<tr>
<td><strong>Expressiveness</strong></td>
<td>One liners, benefit of multiple dispatch</td>
</tr>
</tbody>
</table>
Julia REPL
Read Evaluate Print Loop

[plutonium Development] $ julia

Documentation: https://docs.julialang.org

Type "?" for help, "]?" for Pkg help.

Version 1.5.2 (2020-09-23)
Official https://julialang.org/ release

julia> 2 + 4
6

julia> print("hello")
hello

julia>
Hello World
Print a string to console

julia> println("Hello, 世界")
Hello, 世界
Variables
Binding values to an identifier

```julia
julia> arthur = 42
42
julia> arthur = "forty two"
"forty two"
```

- Reassign to value of different type
  ```julia
  julia> ΔεϕΣ = true
  true
  ```

- Greek letters
  ```julia
  julia> 안녕하세요 = 0.42
  0.42
  ```

- Even Chinese!
String Interpolation and Concatenation

```
julia> engine = "RD-180";
julia> company = "Energomash";
julia> thrust = 3830;
julia> string("The ", company, ", ", engine, ", rocket engine produces ", thrust, ", kN of thrust")
"The Energomash RD-180 rocket engine produces 3830 kN of thrust"
```

Concatenation

"The \ company \ engine \ rocket engine produces \ thrust \ kN of thrust"

Interpolation

"The \ company \ engine \ rocket engine produces \ thrust \ kN of thrust"
## Composite Types

Defining a type made up of multiple parts

### Python

```python
class Knight:
    def __init__(self, name, health, armor):
        self.name = name
        self.health = health
        self.armor = armor
```

### C/C++

```c
struct Knight {
    string name;
    int health;
    int armor;
};
```

### Julia

```julia
struct Knight
    name::String
    health::Int
    armor::Int
end
```

### REPL

```
julia> white = Knight("Sir Lancelot", 6, 2)
Knight("Sir Lancelot", 6, 2)

julia> white.health
6
```
Field Access
Accessing elements in a struct

```julia
struct Knight
    name::String
    health::Int
    armor::Int
end

julia> black = Knight("Sir Morien", 6, 2)
Knight("Sir Morien", 6, 2)

julia> black.name
"Sir Morien"

julia> getfield(black, :name)
"Sir Morien"

julia> getfield(black, 3)
2
```
For Loops

Variations

```python
for x in [3, 4, 5]
    total += x
end
```

```python
for x in 3:5
    total += x
end
```

```python
range = 3:5
for x in range
    total += x
end
```

```python
sum([3, 4, 5])
sum(3:5)
```
Variations

i = 1
while i <= 3
    total += numbers[i]
i += 1
end

i = 3
while 1 <= i <= 3
    total += numbers[i]
i += 1
end

i = 3
while 1 ≤ i ≤ 3
    total += numbers[i]
i += 1
end
If Statement
Variations

```bash
if x > 5
    "large"
elseif x > 3
    "medium"
else
    "small"
end

s = if x > 1000
    "large"
else
    "small"
end

s = x > 1000 ? "large" : "small"
```
Functions
Different ways of defining functions

One-Liner

\[ f(x) = 2x + 4 \]

Multiline with Type Annotations

```julia
function add(x::Int, y::Int)
    return x + y
end
```

```
julia> f(3)
10

julia> add(3, 4)
7
```
Arrays
Arrays

Working with data in tables collectively

<table>
<thead>
<tr>
<th>Amount</th>
<th>Unit Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

2D Array

<table>
<thead>
<tr>
<th>Amount</th>
<th>Unit Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

1D Array

\[
\begin{array}{c}
27 \\
60
\end{array}
\]
Arrays

Working with data in tables collectively

1D Array
## Arrays

Working with data in tables collectively

<table>
<thead>
<tr>
<th>Amount</th>
<th>Unit Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

\[
julia> \text{unitcosts} = [6, 4, 3, 2, 1]
\]

5-element Array{Int64,1}:

\[
6 \\
4 \\
3 \\
2 \\
1
\]

1D Array
# Arrays

Working with data in tables collectively

## 1D Array

<table>
<thead>
<tr>
<th>Amount</th>
<th>Unit Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

\[
\text{amounts} \times \text{unitcosts} = \begin{array}{c} 12 \\ 12 \\ 12 \\ 12 \\ 12 \end{array}
\]

```julia
julia> amounts .* unitcosts
5-element Array{Int64,1}:
    12
    12
    12
    12
    12
```
Arrays

Working with data in tables collectively

2D Array

<table>
<thead>
<tr>
<th>Amount</th>
<th>Unit Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

27  60

Julia>

```
table = [2 6 12;
         3 4 12;
         6 2 12;
         12 1 12]
```

4x3 Array{Int64,2}:

```
2  6  12
3  4  12
6  2  12
12 1  12
```
Vector Slicing

**element**

```
v[2]
```

```
v[1:end]
```

**slice**

```
v[2:3]
```

```
v[:]
```
Matrix Slicing

**element**

\[ A[2, 3] \]

**slice**


**row**

\[ A[2, :] \]

**column**

\[ A[:, 3] \]
Dictionaries
Creating a Dictionary
Different ways of creating dictionaries

```python
d = Dict("two" => 2, "four" => 4)
pairs = ["two" => 2, "four" => 4]
Dict(pairs)
tuples = ["two", 2, "four", 4]
Dict(tuples)
words = ["two", "four"]
nums = [2, 4]
Dict(zip(words, nums))
```
Accessing Elements
Setting and getting dictionary values by key

```
julia> d = Dict("two" => 2, "four" => 4)
Dict{String,Int64} with 2 entries
  "two"  => 2
  "four" => 4

julia> d["two"]
2

julia> d["five"] = 5
5
```

```
julia> d
Dict{String,Int64} with 3 entries:
  "two"  => 2
  "four" => 4
  "five" => 5
```
Functional
Anonymous Functions and Closures

Why are anonymous functions handy?

One-Liner Inlined

```julia
map(x->x^2, [2, 3, 4])
```

Multi-Liner Inlined

```julia
map([2, 3, 4]) do x
    x^2
end
```
Partial Application
Creating new functions by only providing some function arguments

Built-in

```julia
julia> findfirst(x->x == 6, [3, 4, 6, 7, 6])
3

julia> findfirst(==(6), [3, 4, 6, 7, 6])
3

julia> filter(>(6), [3, 4, 6, 7, 6])
3-element Array{Int64,1}:
   6
   7
   6
```

Define Your Own

```julia
import Base: >, <

>(y) = x -> x > y
<(y) = x -> x < y

julia> findfirst(>(6), [3, 4, 6, 7, 6])
4

julia> filter(<(6), [3, 4, 6, 7, 6])
2-element Array{Int64,1}:
   3
   4
```
<table>
<thead>
<tr>
<th>Section</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Tour</td>
<td>Functions, variables, loops, if-statements, arrays</td>
</tr>
<tr>
<td>Programming Language Trade-Offs</td>
<td>Why are dynamic languages slow? Boxing, memory fragmentation</td>
</tr>
<tr>
<td>What is the Secret?</td>
<td>Just in time compilation? Language Design?</td>
</tr>
<tr>
<td>JIT Code Generation</td>
<td>Vector dot product, lowering, abstract syntax tree, LLVM bitcode, native assembly</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>One liners, benefit of multiple dispatch</td>
</tr>
</tbody>
</table>
Boxing and Cache
Boxing

Dynamic languages need to box every value

Boxed Value

We don't know size of data

Actual data such as integer or string

Unboxed Value
Overhead from Classes
Comparing Java and Julia objects

class Point
    int x;
    int y;
end

struct Point
    x::Int
    y::Int
end

Mark
Type
Alignment

member variables

x
y
Memory Fragmentation

With composite types boxing causes even more fragmentation

class Rect
  Point min;
  Point max;
end

struct Rect
  min::Point
  max::Point
end

[Diagram of Rect and Point structures]
Fragmentation of Arrays in Memory

Boxing problems grows with arrays

Java

```
Point[] points = new Point[3];
```

Fragmented Memory

Julia

```
points = Vector{Point}(undef, 3)
```

Contiguous memory
JIT Unfriendly
Julia Version
Which executes fast

```julia
struct Vector2D
    x::Float64
    y::Float64
end

function multiply(u::Vector2D, coeff::Number)
    Vector2D(coeff * u.x, coeff * u.y)
end

julia> v = Vector2D(3, 4)
Vector2D(3.0, 4.0)

julia> multiply(v, 2.0)
Vector2D(6.0, 8.0)
```
function multiply(u, coeff)
    ux = getfield(u, :x)
    if !isa(ux, Float64)
        error("x must be a float")
    end

    uy = getfield(u, :y)
    if !isa(uy, Float64)
        error("y must be a float")
    end

    if coeff isa Int
        k = convert(Int, coeff)
        return Vector2D(coeff * ux, coeff * uy)
    elseif coeff isa Float64
        k = convert(Float64, coeff)
        return Vector2D(k * ux, k * uy)
    else
        error("Unknown type")
    end
end
function multiply(u, coeff)
  ux = getfield(u, :x)
  if ! isa(ux, Float64)
    error("x must be a float")
  end

  uy = getfield(u, :y)
  if ! isa(uy, Float64)
    error("y must be a float")
  end

  if coeff isa Int
    k = convert(Int, coeff)
    return Vector2D(coeff * ux, coeff * uy)
  elseif coeff isa Float64
    k = convert(Float64, coeff)
    return Vector2D(k * ux, k * uy)
  else
    error("Unknown type")
  end
end

Dynamic Version

Slow version

- Dictionary lookup of each member
function multiply(u, coeff)
    ux = getfield(u, :x)
    if !isa(ux, Float64)
        error("x must be a float")
    end

    uy = getfield(u, :y)
    if !isa(uy, Float64)
        error("y must be a float")
    end

    if coeff isa Int
        k = convert(Int, coeff)
        return Vector2D(coeff * ux, coeff * uy)
    elseif coeff isa Float64
        k = convert(Float64, coeff)
        return Vector2D(k * ux, k * uy)
    else
        error("Unknown type")
    end
end
function multiply(u, coeff)
    ux = getfield(u, :x)
    if !isa(ux, Float64)
        error("x must be a float")
    end

    uy = getfield(u, :y)
    if !isa(uy, Float64)
        error("y must be a float")
    end

    if coeff isa Int
        k = convert(Int, coeff)
        return Vector2D(coeff * ux, coeff * uy)
    elseif coeff isa Float64
        k = convert(Float64, coeff)
        return Vector2D(k * ux, k * uy)
    else
        error("Unknown type")
    end
end

Dynamic Version

Slow version

• Dictionary lookup of each member

• Check type of each member

• Coefficient type determination and conversion
**Types Change**

At any time an object could change which members it has and their type.

<table>
<thead>
<tr>
<th>Vector2D</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>values</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Float64</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>Float64</td>
<td></td>
</tr>
</tbody>
</table>

Type of field changed

<table>
<thead>
<tr>
<th>Vector2D</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>values</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Float64</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

Fields get removed or added

<table>
<thead>
<tr>
<th>Vector2D</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>values</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>Float64</td>
<td></td>
</tr>
</tbody>
</table>
Language Tour
Functions, variables, loops, if-statements, arrays

Programming Language Trade-Offs
Why are dynamic languages slow? Boxing, memory fragmentation

What is the Secret?
Just in time compilation? Language Design?

JIT Code Generation
Vector dot product, lowering, abstract syntax tree, LLVM bitcode, native assembly

Expressiveness
One liners, benefit of multiple dispatch
Is it Just in Time Compilation?

Is it the utilization of LLVM which gives Julia its performance?

- **Used same JIT technique in Python?**
  - Don't have to learn new language

- **PyPy**
  - A tracing JIT compiler for all of Python

- **Numba**
  - LLVM based JIT for decorated functions
Does decorating our variables with some beautiful types boost performance?

Type Annotations

- Give hints to compiler about types
- Fixing or limiting type of an argument
What is the Secret Sauce?
Language Design

Design of the language matter more than technology

- Designed from the ground up for LLVM
- Programming style and idioms
  - Standard library
- Multiple Dispatch
Utility of multiple dispatch

```haskell
mutable struct Archer
    health :: Int
end

mutable struct Pikeman
    health :: Int
end

mutable struct Knight
    health :: Int
end

- Archer beats pikeman
- Knight beats archer
- Pikeman beats knight
```
Making Archers, Pikeman and Knights Fight

How the code we are going to write will work

- Units deal damage to each other when fighting
- When health reaches zero, print out who won

```julia
julia> pikeman = Pikeman(5);
julia> archer = Archer(4);
julia> knight = Knight(6);

julia> attack!(archer, pikeman)
Archer killed pikeman

julia> attack!(archer, knight)
Knight killed archer
```
Function `attack!(a::Archer, b::Pikeman)`

```plaintext
b.health -= 4
if b.health <= 0
    println("Archer killed pikeman")
end
end
```

Function `attack!(a::Archer, b::Knight)`

```plaintext
b.health -= 2
if b.health <= 0
    println("Archer killed knight")
    return
end

a.health -= 6
if a.health <= 0
    println("Knight killed archer")
end
end
```

Function `attack!(a::Archer, b::Archer)`

```plaintext
a.health -= 2
b.health -= 2
if a.health <= 0 && b.health <= 0
    println("Archers killed each other")
elseif a.health <= 0 || b.health <= 0
    println("One archer was killed")
end
end
```
Pikeman vs Everybody Else
Utility of multiple dispatch

```
attack!(a::Pikeman, b::Archer) = attack!(b, a)

function attack!(a::Pikeman, b::Knight)
    b.health -= 4
    if a.health <= 0
        println("Pikeman killed cavalry")
    end
end

function attack!(a::Pikeman, b::Pikeman)
    a.health -= 4
    b.health -= 4
    if a.health <= 0 && b.health <= 0
        println("Pikemen killed each other")
    elseif a.health <= 0 || b.health <= 0
        println("One pikeman was killed")
    end
end
```
Single vs Multiple Dispatch

How is what Julia is doing different from what object oriented-languages do?

- Intersection of two circles
- Union of a circle and a square
- Difference of a hexagon and a square

```plaintext
function intersect(c1::Circle, c2::Circle)
    ...
end

function intersect(c::Circle, s::Square)
    ...
end
```
Dynamic Single Dispatch

How a method call is performed in a dynamically typed object-oriented language

1. `intersect(hexagon, shape)`
2. `hexagon.intersect(shape)`

Object

- isa
- position
- side

Hexagon

- super
- union(shape)
- inside(point)
- difference(shape)
- area()

Polygon

- intersect(shape)

Abstract Syntax Tree

lookup(“intersect”)
Static Single Dispatch
How a method call is performed in a statically typed object-oriented language

1. intersect(hexagon, shape)
   hexagon.intersect(shape)

2. object
   - vtable
   - position
   - side

3. methods
   - union(shape)
   - inside(point)
   - difference(shape)
   - area()
   - intersect(shape)

   01101101001
Multiple Dispatch

How Julia does a method lookup at runtime

1. `intersect(hexagon, shape)`

2. **functions**
   - union
   - intersect
   - difference
   - area
   - inside

3. **methods**
   - (Circle, Circle)
   - (Circle, Square)
   - (Hexagon, Square)
   - (Hexagon, Circle)
   - (Shape, Shape)

4. **Machine Code**
   - 01101101001

5. **Abstract Syntax Tree**
Language Tour
Functions, variables, loops, if-statements, arrays

Programming Language Trade-Offs
Why are dynamic languages slow? Boxing, memory fragmentation

What is the Secret?
Just in time compilation? Language Design?

JIT Code Generation
Vector dot product, lowering, abstract syntax tree, LLVM bitcode, native assembly

Expressiveness
One liners, benefit of multiple dispatch
Simulate Julia JIT
Simulate JIT in Julia

How code for addition is generated

```plaintext
add(x::Int, y::Int) = x + y
vaddsd(x::Float64, y::Float64) = x + y
vcvtsi2sd(x::Int) = float(x)
⊕(x::Int, y::Int) = add(x, y)
⊕(x::Float64, y::Float64) = vaddsd(x, y)
⊕(x::Int, y::Float64) = vaddsd(vcvtsi2sd(x), y)
⊕(x::Float64, y::Int) = y ⊕ x
```

REPL

```plaintext
julia> ⊕(2, 3)
5

julia> 3 ⊕ 4
7

julia> 3 ⊕ 4.3
7.3

julia> @code_lowered 3 ⊕ 4
CodeInfo(
  1 - %1 = Main.add(x, y)
  ⊕ return %1
)
```
Simulate JIT in Julia

How code for addition is generated

```
add(x::Int, y::Int) = x+y
vaddsd(x::Float64,y::Float64) = x+y
vcvtsi2sd(x::Int) = float(x)
⊕(x::Int, y::Int) = add(x, y)
⊕(x::Float64, y::Float64) = vaddsd(x, y)
⊕(x::Int, y::Float64) = vaddsd(vcvtsi2sd(x), y)
⊕(x::Float64, y::Int) = y ⊕ x
```

**REPL**

```
julia> @code_lowered 2 ⊕ 2.5
CodeInfo(
  1 - %1 = Main.vcvtsi2sd(x)
  |  %2 = Main.vaddsd(%1, y)
  |    return %2
)

julia> @code_lowered 2.1 ⊕ 2.1
CodeInfo(
  1 - %1 = Main.vaddsd(x, y)
  |    return %1
)
```
Simulate JIT in Julia

How code for addition is generated

```
julia> methods(⊕)
# 4 methods for generic function "⊕":
[1] ⊕(x::Float64, y::Int64)
[2] ⊕(x::Int64, y::Float64)
[3] ⊕(x::Float64, y::Float64)
[4] ⊕(x::Int64, y::Int64)
```
How code for addition is generated

\[
f(a,b) = a + b
\]

```julia
julia> @code_native f(2, 3)
leaq (%rdi,%rsi), %rax
retq

julia> @code_native f(1.0, 3.0)
vaddsd %xmm1, %xmm0, %xmm0
retq

julia> @code_native f(1.0, 3)
vcvtsi2sdq %rdi, %xmm1, %xmm1
vaddsd %xmm0, %xmm1, %xmm0
retq
```
Adding more complicated data types

```julia
struct Vector2D{T <: Number}
    x::T
    y::T
end

function ⊕(u::Vector2D, v::Vector2D)
    Vector2D(u.x ⊕ v.x, u.y ⊕ v.y)
end

function ⊕(u::Vector2D, k::Number)
    Vector2D(u.x ⊕ k, u.y ⊕ k)
end
```

```
julia> u = Vector2D(3, 4)
Vector2D{Int64}(3, 4)

julia> v = Vector2D(1.0, 2.0)
Vector2D{Float64}(1.0, 2.0)

julia> u ⊕ u
Vector2D{Int64}(6, 8)

julia> u ⊕ v
Vector2D{Float64}(4.0, 6.0)

julia> u ⊕ 10
Vector2D{Int64}(13, 14)
```
Expand ⊕ Operator

Adding more complicated data types

```julia
julia> methods(⊕)
# 4 methods for generic function "⊕":
[1] ⊕(x::Float64, y::Int64)
[2] ⊕(x::Int64, y::Float64)
[3] ⊕(x::Float64, y::Float64)
[4] ⊕(x::Int64, y::Int64)
[5] ⊕(u::Vector2D, v::Vector2D)
[6] ⊕(u::Vector2D, k::Number)
```
JIT Magic
**JIT Magic**

Amazing ability of Julia JIT to simplify

\[ \text{bar}(x) = 2x + 3x \]

```julia
function foo(xs...)
    ys = map(xs) do x
        T = typeof(x)
        k = convert(T, 2)
        c = convert(T, 3)
        k*x + c*x
    end
    sum(ys)
end
```

```
julia> bar(1)
5
julia> bar(2)
10
julia> foo(1)
5
julia> foo(2, 1)
15
```

```
julia> @code_llvm bar(7)
%1 =
    mul i64 %0, 5
ret i64 %1

julia> @code_native bar(7)
leaq (%rdi,%rdi,4), %rax
retq
```

```
2x + 3x = 1x + 4x
rax = rdi + 4rdi
```

```
julia> @code_llvm foo(7)
leaq (%rdi,%rdi,4), %rax
retq

julia> @code_native foo(7)
leaq (%rdi,%rdi,4), %rax
retq

julia> @code_llvm foo(2, 1)
leaq (%rdi,%rdi,4), %rax
addq %rsi, %rdi
leaq (%rdi,%rdi,4), %rax
retq
```

```
julia> @code_native foo(2, 1)
leaq (%rdi,%rdi,4), %rax
addq %rsi, %rdi
leaq (%rdi,%rdi,4), %rax
retq
```
**JIT Magic**

Amazing ability of Julia JIT to simplify

```julia
function foo(xs...)  
    ys = map(xs) do x  
        T = typeof(x)  
        k = convert(T, 2)  
        c = convert(T, 3)  
        k*x + c*x  
    end  
    sum(ys)  
end
```

```julia
julia> @code_native foo(2, 1)
addq %rsi, %rdi
leaq (%rdi,%rdi,4), %rax
retq
```

**What is LLVM doing?**

```julia
[2z + 3z, 2w + 3w] = map([z, w]) do x  
    2x + 3x  
end  
sum([2z + 3z, 2w + 3w])
```

**Rearrange and simplify**

2(z+w) + 3(z+w)

1(z+w) + 4(z+w)

**Simplify further**

x = z + w
x + 4x
Language Tour
Functions, variables, loops, if-statements, arrays

Programming Language Trade-Offs
Why are dynamic languages slow? Boxing, memory fragmentation

What is the Secret?
Just in time compilation? Language Design?

JIT Code Generation
Vector dot product, lowering, abstract syntax tree, LLVM bitcode, native assembly

Expressiveness
One liners, benefit of multiple dispatch
One Liners
Toy examples of expressiveness

- Snake case to camel case
- XYZ coordinates from string
- Factorial of five
- Colon separate values

```julia
julia> join(uppercasefirst.(split("how_are_you", '_'))) "HowAreYou"

julia> x, y, z = parse.(Int, split("10 20 30"))
3-element Array{Int64,1}:
  10
  20
  30

julia> y
20

julia> factorial(5)
120

julia> reduce(*, 1:5)
120

julia> join(string.([3, 2, 8]), "::") "3:2:8"
```
Meta Programming
Reducing boilerplate through code generation

```plaintext
mutable struct Archer <: Soldier
    health::Int
    damage::Int
end

mutable struct Pikeman <: Soldier
    health::Int
    damage::Int
end

mutable struct Knight <: Soldier
    health::Int
    damage::Int
end

for T in [:Archer, :Pikeman, :Knight]
    @eval mutable struct $T <: Soldier
        health::Int
        damage::Int
    end
end
```
Thank you!

Erik Engheim
@erikengheim

http://sixty-north.com/blog/post-permalink