German Centre for Integrative Biodiversity Research (iDiv)
Halle-Jena-Leipzig

HELMHOLTZ
Zentrum für Umweltforschung

Programming in Julia

language philosophy and design

Daniel Vedder

iDiv is a research centre of the
Deutsche Forschungsgemeinschaft

Part I

Julia versus other languages

How does a computer work?

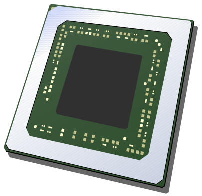
```julia

"""
    mutate!(traits, settings, locivar)

Loop over an array of traits, mutating each value in place along a normal distribution.
`locivar` can be used to scale the variance of the normal distribution used to draw new
trait values (together with `settings[phylconstr]`).
"""
function mutate!(traits::Array{Trait, 1}, settings::Dict{String, Any}, locivar::Float64 = 1.0)
    settings["phylconstr"] * locivar == 0 && return
    for trait in traits
        traitname = settings["traitnames"][trait.nameindex]
        occursin("seqsimilarity", traitname) && settings["fixtol"] && continue
        oldvalue = trait.value
        occursin("tempopt", traitname) && (oldvalue -= 273)
        while oldvalue <= 0 # make sure sd of Normal dist != 0
            oldvalue = abs(rand(Normal(0,0.01)))
        end
        newvalue = rand(Normal(oldvalue, oldvalue * settings["phylconstr"] * locivar))
        (newvalue > 1 && occursin("prob", traitname)) && (newvalue=1.0)
        while newvalue <= 0
            newvalue = rand(Normal(oldvalue, oldvalue * settings["phylconstr"] * locivar))
        end
        occursin("tempopt", traitname) && (newvalue += 273)
        trait.value = newvalue
    end
end

```

High-level language

compilation / interpretation

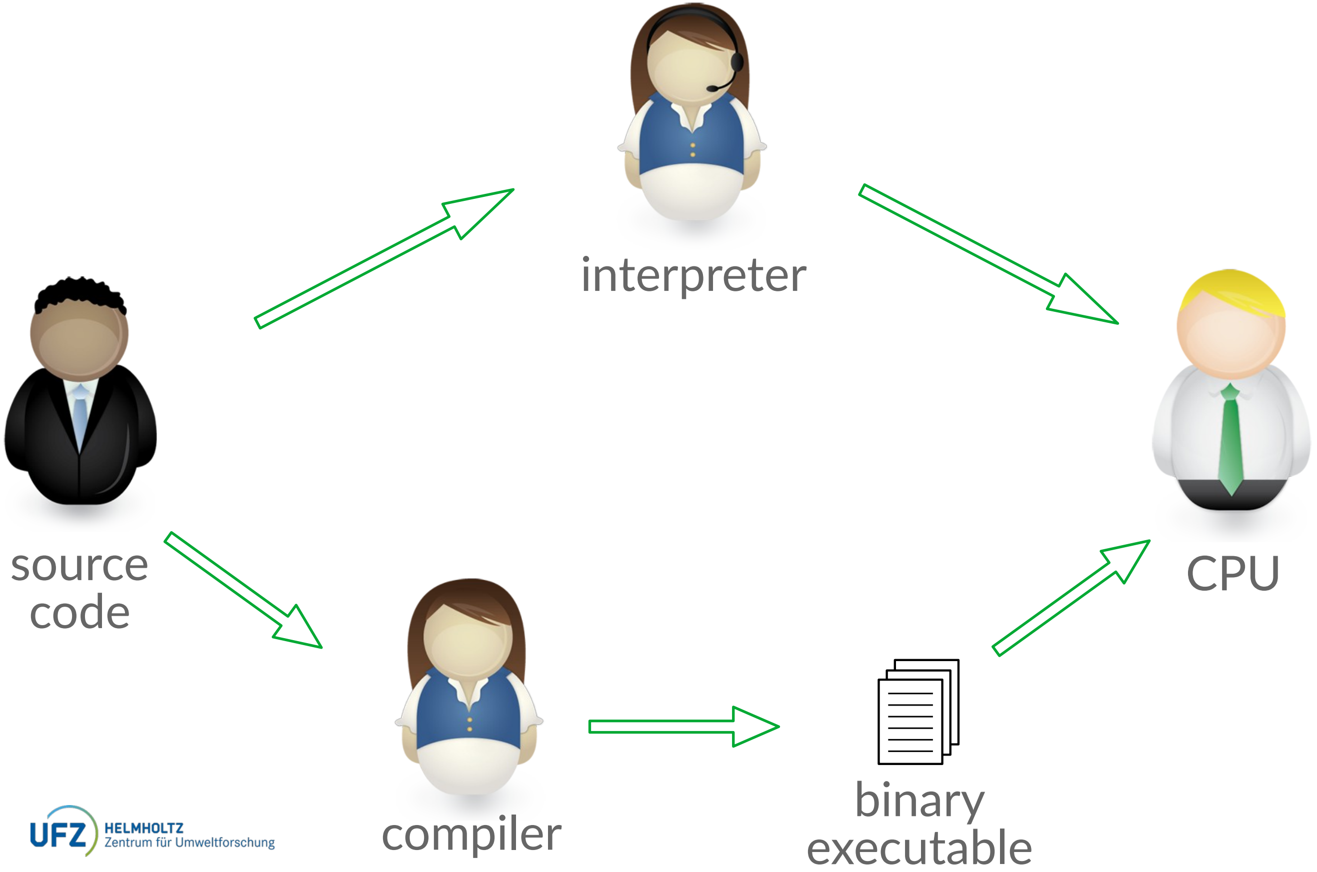
```
FE30- 20 B4 FC 90 F7 60 B1 3C
*FDEDL

FDED-   6C 36 00    JMP   ($0036)
FDF0-   C9 A0       CMP   #$A0
FDF2-   90 02       BCC   $FDF6
FDF4-   25 32       AND   $32
FDF6-   84 35       STY   $35
FDF8-   48          PHA
FDF9-   20 78 FB    JSR   $FB78
FDFC-   68          PLA
FDFD-   A4 35       LDY   $35
FDFF-   60          RTS
FE00-   C6 34       DEC   $34
FE02-   F0 9F       BEQ   $FDA3
FE04-   CA          DEX
FE05-   D0 16       BNE   $FE1D
FE07-   C9 BA       CMP   #$BA
FE09-   D0 BB       BNE   $FDC6
```

assembly



binary electric signals

Compilation vs. interpretation



A closer look at: R

- **Purpose:** *data analysis & visualisation*
- **Context:**
  - Users are scientists, not software developers
  - Most programs are small; apply pre-existing functions to new data
  - Programs are often not specified ahead of time → **exploratory programming**
- **Design choices:**
  - Strongly interactive (REPL) → interpreted
  - Huge library ecosystem for common functionality
  - Dynamic typing (easier for users)

A closer look at: Python

- **Purpose:** *general purpose, especially scripting*
- **Context:**
  - Should be quick & easy to write
  - Should be useful for as many use-cases as possible (both scripts and larger programs)
  - Performance is generally not a priority (except in special libraries)
- **Design choices:**
  - Clean, minimalist syntax
  - Portability, dynamic typing → interpreted
  - Large standard library for common functionality ("batteries included")

A closer look at: C++

- **Purpose:** *general purpose, especially application and systems programming*
- **Context:**
  - Users are expert software developers
  - Often needs to interface directly with hardware, performance is critical
  - Efficiency is more important than ease of use
  - Programs can be very large and complex → **architectural programming**
- **Design choices:**
  - Performance needed → compiled
  - Close to hardware → static typing, no automatic memory management
  - 3rd-party libraries needed for most functionality

iDiv UFZ HELMHOLTZ Zentrum für Umweltforschung

A closer look at: Julia

- **Purpose:** *scientific computing*
- **Context:**
  - Users are not all expert software engineers, but have to carry out complex and intensive calculations → should be easy to use, yet offer high performance
  - Programs are usually novel and can be complex → either exploratory or architectural programming
- **Design choices:**
  - Performance + interactivity → just-in-time compilation
  - Language designed to allow building complex scientific software
  - ***Ideal*: numeric utility of R + simplicity of Python + performance of C++**

Pros and cons of Julia

Advantages

- Syntax is easy to learn and understand
- Performance is comparable to C/C++
- Designed by and for computational scientists → scientific libraries
- Good documentation, helpful community (Discourse forum)
- Flexible & expressive language (multiple dispatch, metaprogramming)

Disadvantages

- Not a typical object-oriented language, need time to learn concepts
- Annoying wait for compilation before every run (TTFX)
- Young language, small community → limited availability of high-quality libraries
- Fast-moving development, future changes may break code

Question Time I

Part II

Julia’s design: the type system

iDiv UFZ HELMHOLTZ Zentrum für Umweltforschung

What are types?

primitive types

(e.g. *int*, *bool*)

composite types

(e.g. *array*, *dict*)

Static vs. dynamic typing

Static typing

- Variable types declared by programmer
- Easier for compiler/interpreter
- Rigid, types have to be known ahead of time and cannot change
- Allows compiler optimisation → better performance
- Mandatory in C/C++, optional in Julia

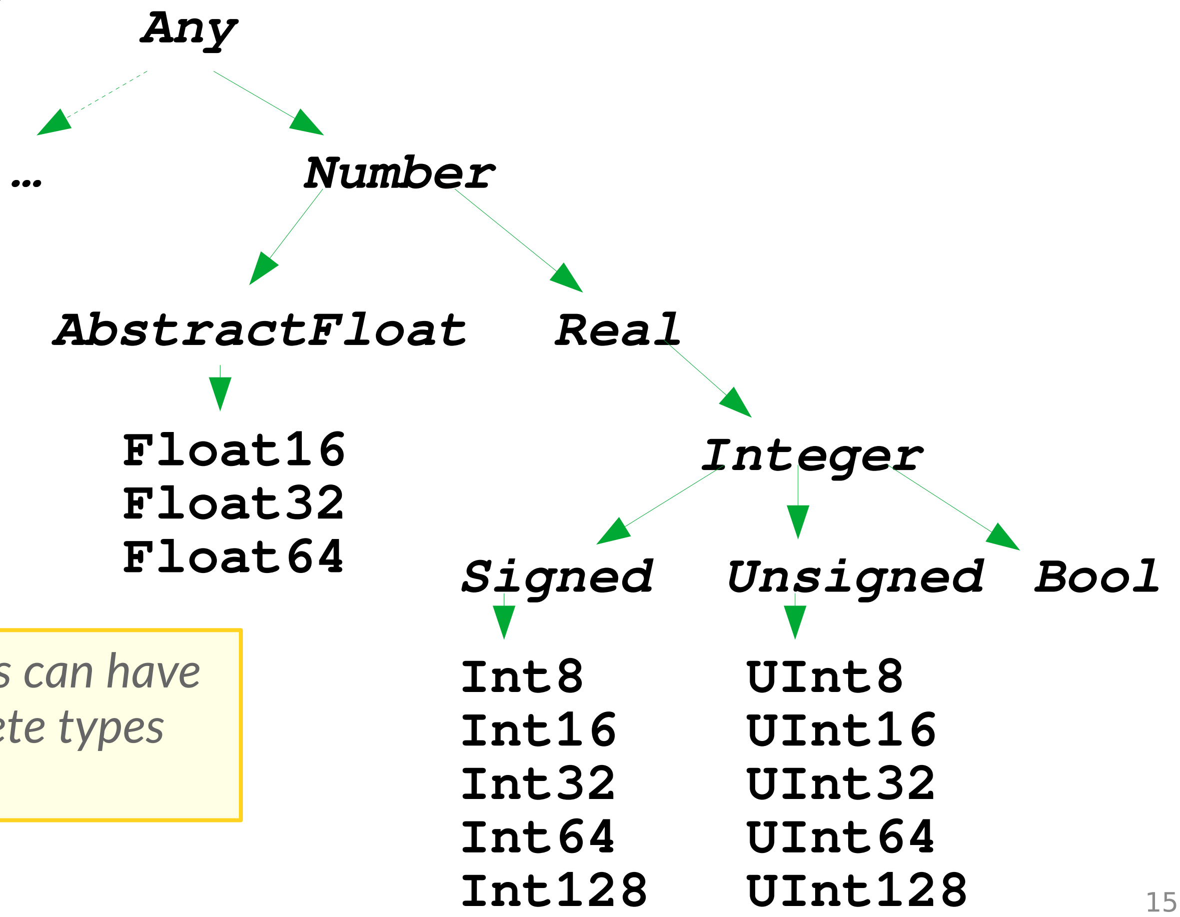
Dynamic typing

- Types inferred automatically by compiler/interpreter
- Easier for programmer
- Flexible, allows types to be determined or change at runtime
- Often slower
- Mandatory in R/Python, default in Julia

OOP the Julian way

- Everything in Julia is an object (even functions!) and every object has a type
- But Julia does not do OOP the way Java/Python/C++ do:
  - In Java/Python/C++, the software is split up into classes, which generate objects that do stuff (classes have methods associated with them)
  - In Julia, the software consists of a collection of functions that act on objects of different types (methods have types associated with them)
    → the focus is on the processes, not the entities!

Type hierarchy



Note: *Only abstract types can have subtypes, but only concrete types can be instantiated.*

iDiv UFZ HELMHOLTZ Zentrum für Umweltforschung

Creating new types: structs

```
julia> mutable struct Foo
           bar::String
           baz::Int
           qux
       end

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

julia> typeof(foo)
Foo

julia> foo.bar
"Hello, world."

julia> foo.baz = 20
20
```

type declaration

object instantiation

only possible with mutable structs!

Dispatching on type: methods

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)

julia> f(2,3)
5

julia> f("hello", "world")
ERROR: MethodError: no method matching +(::String, ::String)

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

julia> f("hello", "world")
"helloworld"
```

***Inheritance:** A method declared for a supertype will automatically apply to all subtypes, unless there are more specialised methods available*

Question Time II

Thank you for your attention!

Now let's get our hands dirty...