nb-l03-julia-types

November 12, 2019

Scientific Computing, TU Berlin, WS 2019/2020, Lecture 03

Jürgen Fuhrmann, WIAS Berlin

1 Additional Info on Julia installation

- There is Julia Pro, a Julia distribution with an additional registry of curated packages It comes bundeled with Juno for the Atom editor
- All info on installation is collected on the course homepage

Thanks Obin Sturm for the hint…

2 Recap

- General info
- Adding packages
- Assignments, simple data types
- Vectors and matrices
- Basic control structures

3 Julia type system

- Julia is a strongly typed language
- Knowledge about the layout of a value in memory is encoded in its type
- Prerequisite for performance
- There are concrete types and abstract types
- See WikiBook for more

3.1 Concrete types

- Every value in Julia has a concrete type
- Concrete types correspond to computer representations of objects
- Inquire type info using `typeof()`
- One can initialize a variable with an explicitely given fixed type
  - Currently posible only in the body of funtions and for return values, not in the global context of Jupyter, REPL

[1]:
```julia
function sometypes()
    i::Int8=10
    @show i,typeof(i)
    x::Float16=5.0
    @show x,typeof(x)
    z::Complex{Float32}=15+3im
    @show z,typeof(z)
    return z
end
z1=sometypes()
@show z1,typeof(z1);
```

```
(i, typeof(i)) = (10, Int8)
(x, typeof(x)) = (Float16(5.0), Float16)
(z, typeof(z)) = (15.0f0 + 3.0f0im, Complex{Float32})
(z1, typeof(z1)) = (15.0f0 + 3.0f0im, Complex{Float32})
```

Vectors and Matrices have concrete types as well:

[2]:
```julia
function sometypesv()
    iv=zeros(Int8, 10)
    @show iv,typeof(iv)
    xv=[Float16(sin(x)) for x in 0:0.1:1]
    @show xv,typeof(xv)
    return xv
end
x1=sometypesv()
@show x1,typeof(x1);
```

```
(iv, typeof(iv)) = (Int8[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], Array{Int8,1})
(xv, typeof(xv)) = (Float16[0.0, 0.09985, 0.1986, 0.2954, 0.3894, 0.4795,
0.5645, 0.644, 0.7173, 0.783, 0.8413], Array{Float16,1})
(x1, typeof(x1)) = (Float16[0.0, 0.09985, 0.1986, 0.2954, 0.3894, 0.4795,
0.5645, 0.644, 0.7173, 0.783, 0.8413], Array{Float16,1})
```

Structs allow to define user defined concrete types

[3]:
```julia
struct Color64
    r::Float64
    g::Float64
    b::Float64
end
c=Color64(0.5,0.5,0.1)
@show c,typeof(c);
```

```
(c, typeof(c)) = (Color64(0.5, 0.5, 0.1), Color64)
```

Types can be parametrized (similar to array)

[4]:
```
struct TColor{T}
    r::T
    g::T
    b::T
end
c=TColor{Float16}(0.5,0.5,0.1)
@show c,typeof(c);
```

```
(c, typeof(c)) = (TColor{Float16}(Float16(0.5), Float16(0.5), Float16(0.1)),
TColor{Float16})
```

3.2 Functions, Methods and Multiple Dispatch

- Functions can have different variants of their implementation depending on the types of parameters passed to them
- These variants are called **methods**
- All methods of a function `f` can be listed calling `methods(f)`
- The act of figuring out which method of a function to call depending on the type of parameters is called **multiple dispatch**

[5]:
```
function test_dispatch(x::Float64)
    println("dispatch: Float64, x=$(x)")
end
function test_dispatch(i::Int64)
    println("dispatch: Int64, i=$(i)")
end
test_dispatch(1.0)
test_dispatch(10)
methods(test_dispatch)
```

```
dispatch: Float64, x=1.0
dispatch: Int64, i=10
```

[5]:
```
# 2 methods for generic function "test_dispatch":
[1] test_dispatch(i::Int64) in Main at In[5]:5
[2] test_dispatch(x::Float64) in Main at In[5]:2
```

- Typically, Julia functions have lots of possible methods
- Each method is compiled to different machine code and can be optimized for the particular parameter types

[6]:
```
using LinearAlgebra
methods(det)
```

[6]:
```
# 23 methods for generic function "det":
[1] det(lu::SuiteSparse.UMFPACK.UmfpackLU{Float64,Int32}) in SuiteSparse.UMFPACK
at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/SuiteSpar
se/src/umfpack.jl:324
```

```
[2] det(lu::SuiteSparse.UMFPACK.UmfpackLU{Complex{Float64},Int32}) in
SuiteSparse.UMFPACK at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/s
tdlib/v1.2/SuiteSparse/src/umfpack.jl:331
[3] det(lu::SuiteSparse.UMFPACK.UmfpackLU{Float64,Int64}) in SuiteSparse.UMFPACK
at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/SuiteSpar
se/src/umfpack.jl:324
[4] det(lu::SuiteSparse.UMFPACK.UmfpackLU{Complex{Float64},Int64}) in
SuiteSparse.UMFPACK at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/s
tdlib/v1.2/SuiteSparse/src/umfpack.jl:331
[5] det(A::SymTridiagonal) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia
-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/tridiag.jl:347
[6] det(A::Tridiagonal) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.
2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/tridiag.jl:627
[7] det(A::UnitUpperTriangular{T,S} where S<:AbstractArray{T,2}) where T in
LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/
v1.2/LinearAlgebra/src/triangular.jl:2492
[8] det(A::UnitLowerTriangular{T,S} where S<:AbstractArray{T,2}) where T in
LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/
v1.2/LinearAlgebra/src/triangular.jl:2493
[9] det(A::UpperTriangular) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/juli
a-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/triangular.jl:2498
[10] det(A::LowerTriangular) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/jul
ia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/triangular.jl:2499
[11] det(A::Symmetric{#s617,S} where S<:(AbstractArray{#s6171,2} where
#s6171<:#s617) where #s617<:Real) in LinearAlgebra at /home/abuild/rpmbuild/BUIL
D/julia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/symmetric.jl:499
[12] det(A::Union{Hermitian{T,S}, Hermitian{Complex{T},S}, Symmetric{T,S}} where
S where T<:Real) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.0/usr
/share/julia/stdlib/v1.2/LinearAlgebra/src/symmetric.jl:498
[13] det(A::Symmetric) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2
.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/symmetric.jl:500
[14] det(D::Diagonal) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.
0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/diagonal.jl:455
[15] det(A::AbstractArray{T,2}) where T in LinearAlgebra at /home/abuild/rpmbuil
d/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/generic.jl:134
1
[16] det(x::Number) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.0/
usr/share/julia/stdlib/v1.2/LinearAlgebra/src/generic.jl:1347
[17] det(A::Eigen) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.0/u
sr/share/julia/stdlib/v1.2/LinearAlgebra/src/eigen.jl:326
[18] det(C::Cholesky) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia-1.2.
0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/cholesky.jl:456
[19] det(C::CholeskyPivoted) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/jul
ia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/cholesky.jl:472
[20] det(F::LU{T,S} where S<:AbstractArray{T,2}) where T in LinearAlgebra at /ho
me/abuild/rpmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/s
rc/lu.jl:378
```

```
[21] det(L::SuiteSparse.CHOLMOD.Factor) in SuiteSparse.CHOLMOD at /home/abuild/r
pmbuild/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/SuiteSparse/src/cholmod.jl
:1773
[22] det(F::Factorization) in LinearAlgebra at /home/abuild/rpmbuild/BUILD/julia
-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/factorization.jl:39
[23] det(J::UniformScaling{T}) where T in LinearAlgebra at /home/abuild/rpmbuild
/BUILD/julia-1.2.0/usr/share/julia/stdlib/v1.2/LinearAlgebra/src/uniformscaling.
jl:185
```

The function/method concept somehow corresponds to C++14 generic lambdas

```
auto myfunc=[](auto  &y, auto &y)
  {
    y=sin(x);
  };
```

is equivalent to

[7]:
```
function myfunc!(y,x)
    y=sin(x)
end
```

[7]:
```
myfunc! (generic function with 1 method)
```

Many generic programming approaches possible in C++ also work in Julia,

If not specified otherwise via parameter types, Julia functions are generic: "automatic auto"

3.3 Abstract types

- Abstract types label concepts which work for a several concrete types without regard to their memory layout etc.
- All variables with concrete types corresponding to a given abstract type (must) share a common interface
- A common interface consists of a set of methods working for all types exhibiting this interface
- The functionality of an abstract type is implicitely characterized by the methods working on it
- "duck typing": use the "duck test" — "If it walks like a duck and it quacks like a duck, then it must be a duck" — to determine if an object can be used for a particular purpose

Examples of abstract types

[8]:
```
function sometypesa()
    i::Integer=10
    @show i,typeof(i)
    x::Real=5.0
    @show x,typeof(x)
    z::Any=15+3im
    @show z,typeof(z)
    return z
```

```
end
sometypesa()
```

```
(i, typeof(i)) = (10, Int64)
(x, typeof(x)) = (5.0, Float64)
(z, typeof(z)) = (15 + 3im, Complex{Int64})
```

[8]: 15 + 3im

Though we try to force the variables to have an abstract type, they end up with having a conrete type which is compatible with the abstract type

3.3.1 The type tree

- Types can have subtypes and a supertype
- Concrete types are the leaves of the resulting type tree
- Supertypes are necessarily abstract
- There is only one supertype for every (abstract or concrete) type:

[9]:
```
supertype(Float64)
```

[9]: AbstractFloat

- Abstract types can have several subtypes

[10]:
```
using InteractiveUtils
subtypes(AbstractFloat)
```

[10]:
```
4-element Array{Any,1}:
 BigFloat
 Float16
 Float32
 Float64
```

- Concrete types have no subtypes

[11]:
```
subtypes(Float64)
```

[11]: 0-element Array{Type,1}

- "Any" is the root of the type tree and has itself as supertype

[12]:
```
supertype(Any)
```

[12]: Any

Walking the the type tree

[13]:
```
function showtypetree(T, level=0)
    println("  " ^ level, T)
    for t in subtypes(T)
        showtypetree(t, level+1)
    end
end
showtypetree(Number)
```

```
Number
  Complex
  Real
    AbstractFloat
      BigFloat
      Float16
      Float32
      Float64
    AbstractIrrational
      Irrational
    Integer
      Bool
      Signed
        BigInt
        Int128
        Int16
        Int32
        Int64
        Int8
      Unsigned
        UInt128
        UInt16
        UInt32
        UInt64
        UInt8
    Rational
```

We can have a nicer walk through the type tree by implementing an interface method `AbstractTrees.children` for types:

[14]:
```
using Pkg
Pkg.add("AbstractTrees")
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo
`https://github.com/JuliaRegistries/General.git`
 Resolving package versions…
  Updating `~/.julia/environments/v1.2/Project.toml`
  [1520ce14] + AbstractTrees v0.2.1
  Updating `~/.julia/environments/v1.2/Manifest.toml`
```

```
  [1520ce14] + AbstractTrees v0.2.1
```

[15]:
```
using AbstractTrees
```

[16]:
```
AbstractTrees.children(x::Type) = subtypes(x)
AbstractTrees.print_tree(Number)
```

```
Number
  Complex
  Real
     AbstractFloat
        BigFloat
        Float16
        Float32
        Float64
     AbstractIrrational
        Irrational
     Integer
        Bool
        Signed
           BigInt
           Int128
           Int16
           Int32
           Int64
           Int8
        Unsigned
           UInt128
           UInt16
           UInt32
           UInt64
           UInt8
     Rational
```

Abstract types are used to dispatch between methods as well

[17]:
```
function test_dispatch(x::AbstractFloat)
    println("dispatch: $(typeof(x)) <:AbstractFloat, x=$(x)")
end
function test_dispatch(i::Integer)
    println("dispatch: $(typeof(i)) <:Integer, i=$(i)")
end
test_dispatch(one(Float16))
test_dispatch(10)
methods(test_dispatch)
```

```
dispatch: Float16 <:AbstractFloat, x=1.0
dispatch: Int64, i=10
```

[17]:
```
# 4 methods for generic function "test_dispatch":
 [1] test_dispatch(i::Int64) in Main at In[5]:5
 [2] test_dispatch(x::Float64) in Main at In[5]:2
 [3] test_dispatch(x::AbstractFloat) in Main at In[17]:2
 [4] test_dispatch(i::Integer) in Main at In[17]:5
```

Now, depending on the input type for `test_dispatch`, a generic or a specific method is called

Testing of type relationships

[18]:
```
@show  Float64<: AbstractFloat
@show  Float64<: Integer
@show  Int16<: AbstractFloat;
```

```
Float64 <: AbstractFloat = true
Float64 <: Integer = false
Int16 <: AbstractFloat = false
```

3.4 The power of multiple dispatch

- Multiple dispatch is one of the defining features of Julia
- Combined with the the hierarchical type system it allows for powerful generic program design
- New datatypes (different kinds of numbers, differently stored arrays/matrices) work with existing code once they implement the same interface as existent ones.
- In some respects C++ comes close to it, but for the price of more and less obvious code

4 Just-in-time compilation and Performance

- Just-in-time compilation is another feature setting Julia apart
- Use the tools from the The LLVM Compiler Infrastructure Project to organize on-the-fly compilation of Julia code to machine code
- Tradeoff: startup time for code execution in interactive situations
- Multiple steps: Parse the code, analyze data types etc.
- Intermediate results can be inspected using a number of macros

From Introduction to Writing High Performance Julia by D. Robinson

4.1 Inspecting the code transformation

Define a function

[19]:
```
g(x)=x+x
@show g(2)
methods(g)
```

```
g(2) = 4
```

[19]:
```
# 1 method for generic function "g":
 [1] g(x) in Main at In[19]:1
```

Parse into abstract syntax tree

[20]:
```julia
@code_lowered g(2)
println("-------------------------------------")
@code_lowered g(2.0)
```

```
-------------------------------------
```

[20]:
```
CodeInfo(
1   %1 = x + x
        return %1
)
```

Type inference according to input

[21]:
```julia
@code_warntype g(2)
println("-------------------------------------")
@code_warntype g(2.0)
```

```
Variables
  #self#::Core.Compiler.Const(g, false)
  x::Int64

Body::Int64
1   %1 = (x + x)::Int64
        return %1
-------------------------------------
Variables
  #self#::Core.Compiler.Const(g, false)
  x::Float64

Body::Float64
1   %1 = (x + x)::Float64
        return %1
```

LLVM Bytecode

[22]:
```julia
@code_llvm g(2)
println("-------------------------------------")
@code_llvm g(2.0)
```

```
;  @ In[19]:1 within `g'
define i64 @julia_g_17078(i64) {
top:
;  @ int.jl:53 within `+'
   %1 = shl i64 %0, 1
;
  ret i64 %1
```

```
}
------------------------------------

;  @ In[19]:1 within `g'
define double @julia_g_17217(double) {
top:
;   @ float.jl:395 within `+'
   %1 = fadd double %0, %0
;
  ret double %1
}
```

Native assembler code

[23]:
```
@code_native g(2)
println("-------------------------------------")
@code_native g(2.0)
```

```
        .text
;   @ In[19]:1 within `g'
;    @ In[19]:1 within `+'
        leaq    (%rdi,%rdi), %rax
;
        retq
        nopw    %cs:(%rax,%rax)
;
------------------------------------
        .text
;   @ In[19]:1 within `g'
;    @ In[19]:1 within `+'
        vaddsd  %xmm0, %xmm0, %xmm0
;
        retq
        nopw    %cs:(%rax,%rax)
;
```

4.2 Performance

Macros for performance testing: - @elapsed: wall clock time used - @allocated: number of allocations - @time: @elapsed and @allocated together - @benchmark: Benchmarking small pieces of code

4.3 Time twice in order to skip compilation time

[24]:
```
function ftest(v::AbstractVector)
    result=0
    for i=1:length(v)
        result=result+v[i]^2
    end
```

```
    return result
end
@time ftest(ones(Float64,100000))
@time ftest(ones(Float64,100000))
```

```
  0.012650 seconds (33.90 k allocations: 2.623 MiB)
  0.000274 seconds (7 allocations: 781.500 KiB)
```

[24]: 100000.0

Run for a different type

[25]:
```
@time ftest(ones(Int64,100000))
@time ftest(ones(Int64,100000))
```

```
  0.009524 seconds (24.03 k allocations: 2.071 MiB)
  0.000098 seconds (7 allocations: 781.500 KiB)
```

[25]: 100000

4.4 Julia performace gotchas:

- Variables changing types
  - Type change assumed to be always possible in global context (outside of a function)
  - Type change due to inconsequential programming
- Memory allocations for intermediate results

4.4.1 Performance in global context

As an exception, for this example we use the CPUTime package which works without macro expansion.

[26]:
```
Pkg.add("CPUTime")
using CPUTime
```

```
 Resolving package versions…
  Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.2/Manifest.toml`
 [no changes]
```

Declare a long vector

[27]:
```
myvec=ones(Float64,1000000);
```

Sum up its values

[28]:
```
CPUtic()
begin
    x=0.0
```

```
    for i=1:length(myvec)
        global x
        x=x+myvec[i]
    end
    @show x
end
CPUtoc();
```

```
x = 1.0e6
elapsed CPU time: 0.229591 seconds
```

Alternatively, put the sum into a function

[29]:
```
function mysum(v)
    x=0.0
    for i=1:length(v)
        x=x+v[i]
    end
    return x
end
```

[29]: `mysum (generic function with 1 method)`

Run again

[30]:
```
CPUtic()
begin
   @show mysum(myvec)
end
CPUtoc();
```

```
mysum(myvec) = 1.0e6
elapsed CPU time: 0.008118 seconds
```

4.5 What happened ?

Julia Gotcha #1: The REPL (terminal) is the Global Scope. - So is the Jupyter notebook - Julia is unable to dispatch on variable types in the global scope as they can change their type anytime - In the global context it has to put all variables into "boxes" allowing to dispatch on ther type at runtime - Avoid this situation by always wrapping your critical code into functions

4.6 Type stability

Use @benchmark for testing small functions

[31]:
```
Pkg.add("BenchmarkTools")
using BenchmarkTools
```

```
 Resolving package versions…
  Updating `~/.julia/environments/v1.2/Project.toml`
```

```
[no changes]
 Updating `~/.julia/environments/v1.2/Manifest.toml`
[no changes]
```

[32]:
```julia
function g()
  x=1
  for i = 1:10
    x = x/2
  end
  return x
end
@benchmark g()
```

[32]:
```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     6.181 ns (0.00% GC)
  median time:      6.265 ns (0.00% GC)
  mean time:        6.572 ns (0.00% GC)
  maximum time:     40.099 ns (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1000
```

[33]:
```julia
function h()
  x=1.0
  for i = 1:10
    x = x/2
  end
  return x
end
@benchmark h()
```

[33]:
```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     1.152 ns (0.00% GC)
  median time:      1.157 ns (0.00% GC)
  mean time:        1.181 ns (0.00% GC)
  maximum time:     29.943 ns (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1000
```

4.7 What happened ?

Gotcha #2: Type instabilities

[34]:
```julia
@code_native g()
```

```
        .text
;  @ In[32]:2 within `g'
        pushq   %rax
        movb    $2, %dl
        movl    $1, %ecx
        movabsq $139985165224504, %rax  # imm = 0x7F50D60C0A38
        vmovsd  (%rax), %xmm0           # xmm0 = mem[0],zero
        movl    $9, %eax
        movabsq $139985165224512, %rsi  # imm = 0x7F50D60C0A40
        vmovsd  (%rsi), %xmm1           # xmm1 = mem[0],zero
        andb    $3, %dl
;  @ In[32]:4 within `g'
        cmpb    $1, %dl
        jne     L83
        jmp     L93
        nopw    %cs:(%rax,%rax)
L64:
        vmovq   %xmm0, %rcx
        addq    $-1, %rax
        movb    $1, %dl
        andb    $3, %dl
        cmpb    $1, %dl
        je      L93
L83:
        cmpb    $2, %dl
        jne     L104
;   @ int.jl:59 within `/'
;    @ float.jl:271 within `float'
;     @ float.jl:256 within `Type' @ float.jl:60
        vcvtsi2sdq      %rcx, %xmm2, %xmm0
;
;   @ float.jl:401 within `/'
L93:
        vmulsd  %xmm1, %xmm0, %xmm0
;
;   @ range.jl:595 within `iterate'
;    @ promotion.jl:403 within `=='
        testq   %rax, %rax
;
        jne     L64
;  @ In[32]:6 within `g'
        popq    %rax
```

```
        retq
;   @ In[32]:4 within `g'
L104:
        movabsq $jl_throw, %rax
        movabsq $jl_system_image_data, %rdi
        callq   *%rax
        nop
;
```

[35]:

```
@code_native h()
```

```
        .text
;   @ In[33]:2 within `h'
        movabsq $139985165224624, %rax  # imm = 0x7F50D60C0AB0
;   @ In[33]:6 within `h'
        vmovsd  (%rax), %xmm0           # xmm0 = mem[0],zero
        retq
        nop
;
```

Once again, "boxing" occurs to handle x: in `g()` it changes its type from Int64 to Float64:

[36]:

```
@code_warntype g()
```

```
Variables
  #self#::Core.Compiler.Const(g, false)
  x::Union{Float64, Int64}
  @_3::Union{Nothing, Tuple{Int64,Int64}}
  i::Int64

Body::Float64
1         (x = 1)
   %2  = (1:10)::Core.Compiler.Const(1:10, false)
         (@_3 = Base.iterate(%2))
   %4  = (@_3::Core.Compiler.Const((1, 1), false) ===
nothing)::Core.Compiler.Const(false, false)
   %5  = Base.not_int(%4)::Core.Compiler.Const(true, false)
        goto #4 if not %5
2  %7  = @_3::Tuple{Int64,Int64}::Tuple{Int64,Int64}
         (i = Core.getfield(%7, 1))
   %9  = Core.getfield(%7, 2)::Int64
         (x = x / 2)
         (@_3 = Base.iterate(%2, %9))
   %12 = (@_3 === nothing)::Bool
   %13 = Base.not_int(%12)::Bool
        goto #4 if not %13
3        goto #2
4        return x::Float64
```

[37]:
```
@code_warntype h()
```

```
Variables
  #self#::Core.Compiler.Const(h, false)
  x::Float64
  @_3::Union{Nothing, Tuple{Int64,Int64}}
  i::Int64

Body::Float64
1         (x = 1.0)
    %2  = (1:10)::Core.Compiler.Const(1:10, false)
          (@_3 = Base.iterate(%2))
    %4  = (@_3::Core.Compiler.Const((1, 1), false) ===
nothing)::Core.Compiler.Const(false, false)
    %5  = Base.not_int(%4)::Core.Compiler.Const(true, false)
         goto #4 if not %5
2   %7  = @_3::Tuple{Int64,Int64}::Tuple{Int64,Int64}
          (i = Core.getfield(%7, 1))
    %9  = Core.getfield(%7, 2)::Int64
          (x = x / 2)
          (@_3 = Base.iterate(%2, %9))
    %12 = (@_3 === nothing)::Bool
    %13 = Base.not_int(%12)::Bool
         goto #4 if not %13
3         goto #2
4         return x
```

So, when in doubt, explicitely declare types of variables

5 Structuring your code: modules, files and packages

- Complex code is split up into several files
- Avoid name clashes for code from different places
- Organize the way to use third party code

5.1 Modules

- Modules allow to encapsulate implementation into different namespaces

[38]:
```
module TestModule
function mtest(x)
    println("mtest: x=$(x)")
end
export mtest
end
```

[38]: `Main.TestModule`

- Module content can be accesse via qualified names

[39]:
```
TestModule.mtest(13)
```

```
mtest: x=13
```

- "using" makes all exported content of a module available without prefixing
- The '.' before the module name refers to local modules defined in the same file

[40]:
```
using .TestModule
mtest(23)
```

```
mtest: x=23
```

5.1.1 Finding modules in the file system

- Put single file modules having the same name as the module into a directory which in on the `LOAD_PATH`
- Call "using" or "import" with the module
- You can modify your `LOAD_PATH` by adding e.g. the actual directory

[41]:
```
push!(LOAD_PATH, pwd())
```

[41]:
```
5-element Array{String,1}:
 "@"
 "@v#.#"
 "@stdlib"
 "/home/fuhrmann/Wias/teach/scicomp/course"
 "/home/fuhrmann/Wias/teach/scicomp/course"
```

Do this e.g. in the startup file `.julia/config/startup.jl`

- Create a module in the file system (normally, use your editor...) (yes, we can have multiline strings and `"` in them with `"""` ...)

[42]:
```
open("TestModule1.jl", "w") do io
    write(io, """
    module TestModule1
    function mtest1(x)
        println("mtest1: x=",x)
    end
    export mtest1
    end
    """)
end
```

[42]:
```
88
```

- Import, enabling qualified access

[43]:
```
using TestModule1
TestModule1.mtest1(23)
```

```
 Info: Recompiling stale cache file
/home/fuhrmann/.julia/compiled/v1.2/TestModule1.ji for TestModule1 [top-level]
 @ Base loading.jl:1240
```

```
mtest1: x=23
```

- Import, enabling unqualified access of

[44]:
```
using TestModule1
mtest1(23)
```

```
mtest1: x=23
```

5.1.2 Packages in the file system

- Packages are found via the same mechanism
- Part of the load path are the directory with downloaded packages and the directory with packages under development
- Each package is a directory named `Package` with a subdirectory `src`
- The file `Package/src/Package.jl` defines a module named `Package`
- More structures in a package:
  - Documentation build recipes
  - Test code
  - Dependency description
  - UUID (Universal unique identifier)
- Default packages (e.g. the package manager Pkg) are always available
- Use the package manager to checkout a new package via the registry

5.1.3 Including code from files

- The include statement allows just to include the code in a given file

[45]:
```
open("myfile.jl", "w") do io
    write(io, """myfiletest(x)=println("myfiletest: x=",x)""")
end
```

[45]: 41

[46]:
```
include("myfile.jl")
myfiletest(23)
```

```
myfiletest: x=23
```

5.1.4 How to return homework

- For homework assignements I want you to write single file modules with a standard structure

[47]:

```julia
module MyHomework
function main(;optional_parameter)
    println("Hello World")
end
end
```

[47]: Main.MyHomework

This notebook was generated using [Literate.jl](https://github.com/fredrikekre/Literate.jl).