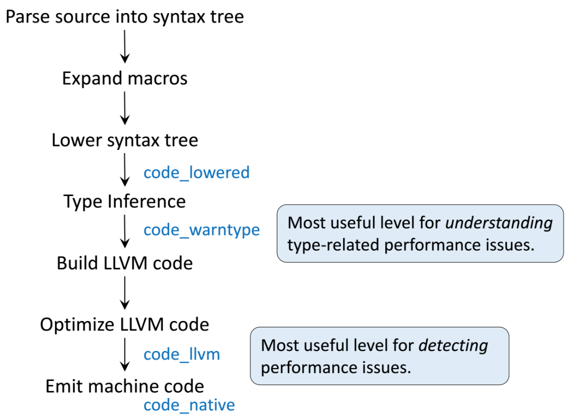
```julia
begin
    using Pkg;
    Pkg.activate(mktempdir());
    Pkg.add(["PlutoUI","BenchmarkTools"]);
    using PlutoUI, BenchmarkTools
end
```

# Julia: just-in-time compilation and Performance

## The JIT

- Just-in-time compilation is another feature setting Julia apart, as it was developed with this possibility in mind.
- Julia uses 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 (blue color in the diagram below)




From **Introduction to Writing High Performance Julia** by D. Robinson

### Let us see what is going on:

```
g (generic function with 1 method)
```

```julia
g(x,y)=x+y
```

- Call with integer parameter:

```
5
```

```julia
g(2,3)
```

- Call with floating point parameter:

```
5.0
```

```julia
g(2.0,3.0)
```

- The macro `@code_lowered` describes the abstract syntax tree behind the code

```
CodeInfo(
1 ─ %1 = x + y
└──      return %1
)
```

```julia
@code_lowered g(2,3)
```

```
CodeInfo(
1 ─ %1 = x + y
└──      return %1
)
```

```julia
@code_lowered g(2.0,3.0)
```

- `@code_warntype` (with output to terminal) provides the result of type inference (detection ot the parameter types and coorsponding choice of the translation strategy) according to the input:

```
Variables
  #self#::Core.Compiler.Const(Main.workspace242.g, false)
  x::Int64
  y::Int64

Body::Int64
1 ─ %1 = (x + y)::Int64
└──      return %1
```

```julia
with_terminal() do
    @code_warntype g(2,3)
end
```

```
Variables
  #self#::Core.Compiler.Const(Main.workspace242.g, false)
  x::Float64
  y::Float64

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

```julia
with_terminal() do
    @code_warntype g(2.0,3.0)
end
```

- `@llvm_bytecode` prints the LLVM intermediate byte code representation:

```
;  @ /home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c
33-7f0c5f3ed551:1 within `g'
define i64 @julia_g_2204(i64, i64) {
top:
; ┌ @ int.jl:86 within `+'
   %2 = add i64 %1, %0
; └
  ret i64 %2
}
```

```julia
with_terminal() do
    @code_llvm g(2,3)
end
```

```
;  @ /home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c
33-7f0c5f3ed551:1 within `g'
define double @julia_g_2206(double, double) {
top:
; ┌ @ float.jl:401 within `+'
```

```
    %2 = fadd double %0, %1
;  └
   ret double %2
}
```

```
with_terminal() do
    @code_llvm g(2.0,3.0)
end
```

- Finally, `@code_native` prints the assembler code generated, which is a close match to the machine code sent to the CPU:

```
	.text
; ┌ @ nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c33-7f0c5f3ed551:1 within `g'
; │┌ @ int.jl:86 within `+'
	leaq	(%rdi,%rsi), %rax
; │└
	retq
	nopw	%cs:(%rax,%rax)
	nop
; └
```

```
with_terminal() do
    @code_native g(2,3)
end
```

```
	.text
; ┌ @ nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c33-7f0c5f3ed551:1 within `g'
; │┌ @ float.jl:401 within `+'
	vaddsd	%xmm1, %xmm0, %xmm0
; │└
	retq
	nopw	%cs:(%rax,%rax)
	nop
; └
```

```
with_terminal() do
    @code_native g(2.0,3.0)
end
```

We see that for the very same function, Julia creates different variants of executable code depending on the data types of the parameters passed. In certain sense, this extends the multiple dispatch paradigm to the lower level by automatically created methods.

# Performance measurment

- Julia provides a number of macros to support performance testing.
- Performance measurement of the first invocation of a function includes the compilation step. If in doubt, measure timing twice.
- Pluto has the nice feature to indicate the execution time used below the lower right corner of a cell. There seems to be also some overhead hidden in the pluto cell handling which is however not measured.

- `@elapsed` : wall clock time used returned as a number.

```
using LinearAlgebra
```

```
f (generic function with 1 method)
```

```
f(n1,n2)= mapreduce(x->norm(x,2),+,[rand(n1) for i=1:n2])
```

```
0.004961619
```

```
@elapsed f(1000,1000)
```

- `@allocated` : sum of memory allocated (including temporary) during the excution of the code. For storing intermediate and final calculation results, computer languages request memory from the operating system. This process is called allocation. Allocations as a rule are linked with lots of bookkeeping, so they can slow down code.

```
8136128
```

```
@allocated f(1000,1000)
```

- `@time` : `@elapsed` and `@allocated` together, with output to the terminal. Be careful to time at least twice in order to take into account compilation time. In addition, the number of allocations is printed along with time spent for garbage collection. Garbage collection is the process of returning unused (temporary) memory to the system.

```
0.004944 seconds (2.00 k allocations: 15.518 MiB)
```

```
with_terminal() do
    @time f(1000,2000)
end
```

- `@benchmark` from `BenchmarkTools.jl` creates a statistic over multiple samples in order to give a more reliable estimate.

```
BenchmarkTools.Trial:
  memory estimate:  7.76 MiB
  allocs estimate:  1001
  --------------
  minimum time:     1.434 ms (0.00% GC)
  median time:      1.564 ms (0.00% GC)
  mean time:        1.922 ms (11.80% GC)
  maximum time:     4.469 ms (35.51% GC)
  --------------
  samples:          2601
  evals/sample:     1
```

```
@benchmark f(1000,1000)
```

# Some performance gotchas

In order to write efficient Julia code, a number recommendations should be followed.

## Gotcha #1: global variables

```
myvec =
 Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1
```

```
myvec=ones(Float64,1_000_000)
```

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

```
0.006184016
```

```
@elapsed mysum(myvec)
```

```
0.115712977
```

```
@elapsed begin
    x=0.0
    for i=1:length(myvec)
        global x
        x=x+myvec[i]
    end
```

```
end
```

- Observation: both the begin/end block and the function do the same operation and calculate the same value. However the function is faster.
- The code within the begin/end clause works in the *global context*, whereas in `myfunc`, it works in the scope of a function. 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" tagged with type information allowing to dispatch on their type at runtime (this is by the way the default mode of Python). In functions, it has a chance to generate specific code for known types.
- This situation als occurs in the REPL.

- Conclusion: **Avoid Julia Gotcha #1 by wrapping time critical code into functions and avoiding the use of global variables.**
- In fact it is anyway good coding style to separate out pieces of code into functions

## Gotcha #2: type instabilities

```
f1 (generic function with 1 method)
```

```
function f1(n)
  x=1
  for i = 1:n
    x = x/2
  end
end
```

```
f2 (generic function with 1 method)
```

```
function f2(n)
  x=1.0
  for i = 1:n
    x = x/2
  end
end
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     5.260 ns (0.00% GC)
  median time:      5.291 ns (0.00% GC)
  mean time:        5.439 ns (0.00% GC)
  maximum time:     32.940 ns (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1000
```

```
@benchmark f1(10)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     1.209 ns (0.00% GC)
  median time:      1.215 ns (0.00% GC)
  mean time:        1.253 ns (0.00% GC)
  maximum time:     28.701 ns (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1000
```

```
@benchmark f2(10)
```

- Observation: function `f2` is faster than `f1` for the same operations.

```
	.text
; ┌ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:1 within `f1'
	pushq	%rax
; │ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:3 within `f1'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:280 within `UnitRange'
; │││┌ @ range.jl:285 within `unitrange_last'
; ││││┌ @ operators.jl:350 within `>='
; │││││┌ @ int.jl:441 within `<='
	testq	%rdi, %rdi
; │└└└└└
```

```
    jle L51
    movq    %rdi, %rax
    sarq    $63, %rax
    andnq   %rdi, %rax, %rax
; │ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:4 within `f1'
    decq    %rax
    movb    $2, %cl
    nopw    (%rax,%rax)
L32:
```

```
with_terminal() do
        @code_native f1(10)
end
```

```
    .text
; ┌ @ nb05-julia-jit.jl#==#36244b3c-01e4-11eb-3828-2fa69b8b0835:4 within `f2'
    retq
    nopw    %cs:(%rax,%rax)
    nopl    (%rax,%rax)
; └
```

```
with_terminal() do
        @code_native f2(10)
end
```

```
Variables
  #self#::Core.Compiler.Const(Main.workspace237.f1, false)
  n::Int64
  x::UNION{FLOAT64, INT64}
  @_4::UNION{NOTHING, TUPLE{INT64,INT64}}
  i::Int64

Body::Nothing
1 ─       (x = 1)
│   %2  = (1:n)::Core.Compiler.PartialStruct(UnitRange{Int64}, Any[Core.Compiler.Const(1, f
alse), Int64])
│         (@_4 = Base.iterate(%2))
│   %4  = (@_4 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_4::Tuple{Int64,Int64}::Tuple{Int64,Int64}
│         (i = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│         (x = x / 2)
│         (@_4 = Base.iterate(%2, %9))
```

```
with_terminal() do
        @code_warntype f1(10)
end
```

```
Variables
  #self#::Core.Compiler.Const(Main.workspace237.f2, false)
  n::Int64
  x::Float64
  @_4::UNION{NOTHING, TUPLE{INT64,INT64}}
  i::Int64

Body::Nothing
1 ─       (x = 1.0)
│   %2  = (1:n)::Core.Compiler.PartialStruct(UnitRange{Int64}, Any[Core.Compiler.Const(1, f
alse), Int64])
│         (@_4 = Base.iterate(%2))
│   %4  = (@_4 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_4::Tuple{Int64,Int64}::Tuple{Int64,Int64}
│         (i = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│         (x = x / 2)
│         (@_4 = Base.iterate(%2, %9))
```

```
with_terminal() do
        @code_warntype f2(10)
end
```

- Once again, "boxing" occurs to handle x: in `g()` it changes its type from Int64 to Float64. We see this with the union type for x in @code_warntype
- Conclusion: **Avoid Julia Gotcha #2 by ensuring variables keep their type also in functions**.

### Gotcha #6: allocations

```
mymat = 10×100000 Array{Float64,2}:
 0.0789994  0.855449  0.564277  0.853326  …  0.524021  0.557067  0.0792928
 0.543264   0.616861  0.21341   0.61336      0.105882  0.980176  0.422125
 0.741373   0.878709  0.78528   0.838547     0.819484  0.859998  0.55475
```

```
0.68516    0.0604356  0.348658  0.776724      0.387086   0.370163    0.12667
0.970368   0.745995   0.72687   0.906995      0.0665305  0.0681725   0.546152
0.684037   0.450483   0.870659  0.79275   …   0.986119   0.206697    0.857506
0.544756   0.754953   0.591328  0.312024      0.785961   0.269252    0.709248
0.783502   0.83581    0.720681  0.960472      0.138532   0.00412415  0.547862
0.169042   0.177096   0.203141  0.619686      0.0963809  0.575215    0.0926853
0.817974   0.691319   0.402212  0.242962      0.774398   0.471102    0.700982
```

```julia
mymat=rand(10,100000)
```

- Define three different ways of summing of squares of matrix rows:

```
g1 (generic function with 1 method)
```

```julia
function g1(a)
    y=0.0
    for j=1:size(a,2)
        for i=1:size(a,1)
            y=y+a[i,j]^2
        end
    end
    y
end
```

```
g2 (generic function with 1 method)
```

```julia
function g2(a)
    y=0.0
    for j=1:size(a,2)
        y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
g3 (generic function with 1 method)
```

```julia
function g3(a)
    y=0.0
    for j=1:size(a,2)
        @views y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
true
```

```julia
g1(mymat)≈ g2(mymat) && g2(mymat)≈ g3(mymat)
```

```
BenchmarkTools.Trial:
  memory estimate:  16 bytes
  allocs estimate:  1
  --------------
  minimum time:     908.531 μs (0.00% GC)
  median time:      987.354 μs (0.00% GC)
  mean time:        1.006 ms (0.00% GC)
  maximum time:     1.785 ms (0.00% GC)
  --------------
  samples:          4967
  evals/sample:     1
```

```julia
@benchmark g1(mymat)
```

```
BenchmarkTools.Trial:
  memory estimate:  15.26 MiB
  allocs estimate:  100001
  --------------
  minimum time:     3.385 ms (0.00% GC)
  median time:      3.610 ms (0.00% GC)
  mean time:        3.851 ms (6.36% GC)
  maximum time:     6.989 ms (12.93% GC)
  --------------
  samples:          1298
  evals/sample:     1
```

```julia
@benchmark g2(mymat)
```

```
BenchmarkTools.Trial:
  memory estimate:  16 bytes
  allocs estimate:  1
  --------------
  minimum time:     793.426 μs (0.00% GC)
  median time:      876.234 μs (0.00% GC)
  mean time:        891.997 μs (0.00% GC)
  maximum time:     1.859 ms (0.00% GC)
  --------------
```

```
samples:          5600
evals/sample:     1
```

```
@benchmark g3(mymat)
```

- Observation: g3 is the fastest implemetation, then comes g1 and then g2.
- The difference between g2 and g1 is that each time we use a matrix slice `a[:,i]`, memory is allocated and data copied. Only then the mapreduce is employed, and the intermediate memory is garbage collected.
- The difference between g2 and g1 lies in the use of the `@views` macro which allows to avoid the creation of intermediae memory for matrix rows.
- Conclusion: avoid **Gotcha #6 by carefully checking your code for allocations** and avoiding the use of temporary memory.