Meta-Programming in Julia

The code writes itself

Lab Workshop – Nicolas Jacquin

Section 1: Why?

Sometimes, we need to work with the code syntax directly, treating it as data. This comes up often for logging, benchmarking and monitoring (think of `@time` or `@btime` from `BenchmarkTools` and `@showprogress` from `ProgressMeter`), adding extra features to existing code without rewriting it (think of `@threads`, `@async`), changing the meaning of some code structures (with @assert to make something a unit test) or cheeky compiler tricks (@turbo, which promises the compiler your code doesn't index out of bounds, iterate over empty collections, order doesn't matter, etc...)

We want to write things like this: `@log_step x + y * z`

And get outputs like this: `[LOG] x + y * z = 42`

This is behaviour that's difficult to emulate with a function. Behold, an abberation:

```
14
```

```julia
begin
    function log_step(expr, value)
        println("[LOG] ", expr, " = ", value)
        value
    end
    a = 2; b = 3; c = 4
    log_step("a + b * c", a + b * c)
end
```

```
[LOG] a + b * c = 14
```

This is worse than useless. I need to pay attention to it every time I refactor the code to make sure the expression stays up to date with what the code represents, meaning the string can lie. An error is pretty much guaranted to happen eventually.

Fortunately, julia allows us to manipulate the code's syntax tree to create macros, such as `@time` or `@assert` to treat code as data. Let's use another macro to take a look at what `@assert`, as in this workshop, we will learn how to do exactely that.

```
md"""
This is worse than useless. I need to pay attention to it every time I refactor the code to make sure the expression stays up to date with what the code represents, meaning the string can lie. An error is pretty much guaranted to happen eventually.

Fortunately, julia allows us to manipulate the code's syntax tree to create macros, such as `@time` or `@assert` to treat code as data. Let's use another macro to take a look at what `@assert`, as in this workshop, we will learn how to do exactely that.
"""
```

```
:(if 1 + 1 == 2
      nothing
  else
      Base.throw(Base.AssertionError("1 + 1 == 2"))
  end)
```

```
@macroexpand(@assert 1+1==2)
```

Error message

```
AssertionError: 1 + 1 == 3
```

Show stack trace...

```
begin
    @assert 1+1 == 2
    @assert 1+1 == 3
end
```

Section 2: Code is data

In julia, code itself is a data structure of type `Expr`. You can explicitely instantiate `Expr` objects using the : (quote) operator. This is called "quoting", it tells julia that this code should not be ran (yet), but represented

```julia
begin
    ex = :(a + b * c)
    println(typeof(ex))
end
```

```
Expr
```

We can use the `dump` function to look at how the `Expr` struct is actually organised, you'll see the code itself is represented within the struct.

```julia
dump(ex)
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Symbol a
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol *
        2: Symbol b
        3: Symbol c
```

To analyse an `Expr`, julia (and many other languages), organises it into a syntax tree, a way to systematically describe the code structure.

```julia
Meta.show_sexpr(ex)
```

```
(:call, :+, :a, (:call, :*, :b, :c))
```

If you're familiar with Lisp, this syntax should be familiar to you. The root node is `:call`, `:+` is and argument and the function to call, `:a` and `(:call, :*, :b, :c)` are arguments to `:+`, with the second one also being an expression by itself. We can isolate these elements specifically

```julia
begin
    println(ex.head)
    println(ex.args)
end
```

```
call
Any[:+, :a, :(b * c)]
```

We can do `Expr` "blocks" to prepare ourselves to write macros

```julia
begin
    blk = quote
        x = 1
        y = 2
        x + y
    end

    dump(blk)
end
```

```
Expr
  head: Symbol block
  args: Array{Any}((6,))
    1: LineNumberNode
      line: Int64 3
      file: Symbol C:\Users\nicol\.julia\pluto_notebooks\Remarkable journal.jl#=
=#4ec8af12-5d0a-4699-9b5e-ffd66c87c52e
    2: Expr
      head: Symbol =
      args: Array{Any}((2,))
        1: Symbol x
        2: Int64 1
    3: LineNumberNode
      line: Int64 4
      file: Symbol C:\Users\nicol\.julia\pluto_notebooks\Remarkable journal.jl#=
=#4ec8af12-5d0a-4699-9b5e-ffd66c87c52e
    4: Expr
      head: Symbol =
      args: Array{Any}((2,))
        1: Symbol y
        2: Int64 2
    5: LineNumberNode
      line: Int64 5
      file: Symbol C:\Users\nicol\.julia\pluto_notebooks\Remarkable journal.jl#=
=#4ec8af12-5d0a-4699-9b5e-ffd66c87c52e
    6: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Symbol x
        3: Symbol y
```

Section 3: Running quoted code

We know we can write code as data, but none of that is useful if we can't *run* said data (code).

- quoting (:) is generating code (an `Expr`)
- evaluating (eval) is running said code

```
begin
    x = 10
    println("quoted code expression: ", :(x + 1))
    println("evaluated code expression: ", eval(:(x + 1)))
end
```

```
quoted code expression: x + 1
evaluated code expression: 11
```

Altough `eval` is usually not something you want inside a macro as it runs in global scope and breaks precompilation

Section 4: Interpolation

We can represent code as `Expr` and run said code, but how do we actually make it useful programatically?

With interpolation, you can insert expressions into quoted code, allowing for a programmatic approach to `Expr` generation

```
begin
    y = :a
    println(:(y + 1))
    println(:($y + 1))
end
```

```
y + 1
a + 1
```

Interpolation is tree splicing (not just string concatenation), which allows you to generate structured code programatically (in a way that's sometimes not possible with functions)

```
Meta.show_sexpr(:($y + 1))
```

```
(:call, :+, :a, 1)
```

When you treat code as data, you can manipulate it in every way you would standard data, weaving whatever you want into the syntax tree the same way you would with actual values

```julia
begin
    fields = [:a, :b, :c]

    exp = quote
        $(fields[1]) + $(fields[2])
    end
    println(exp)
end
```

```
begin
    #= C:\Users\nicol\.julia\pluto_notebooks\Remarkable journal.jl#==#0c794ddf-722
e-4dcf-a27d-4b30ad90ba18:5 =#
    a + b
end
```

You can even do nested interpolation expression building

```julia
begin
    println(:( $(Expr(:call, :+, :a, :b)) * c ))
end
```

```
(a + b) * c
```

Section 5 - Macro Building

now that we know how to generate live code programatically, we can look at practical examples of macro building. Weaving around the syntax tree of our code to add useful bits.

Let's write a macro that adds timing around a block (essentially `@time`)

```
@my_time (macro with 1 method)
```

```julia
macro my_time(ex)
    quote
        t0 = time()
        result = $(ex)
        println("Elapsed: ", time() - t0)
        result
    end
end
```

```
499853.19616667007
```

```julia
@my_time sum(rand(10^6))
```

```
Elapsed: 0.20000004768371582
```

using the `macro` block, the quoted code will be evaluated when the macro is executed, and the `@my_macro` syntax can be used, where the first parameter of the macro is the `Expr` the macro is executed on

However, unlike a function, by default a macro does not have its own scope. This means we can create variable capture issues.

```julia
begin
    t0 = 100
    println(@my_time t0 + 1)
end
```

```
Elapsed: 0.0
1.766150691938e9
```

It is however quite easy to fix. `esc` tells julia "this code belongs to the caller, not the macro", essentially separating the macro's scope from the `Expr` passed to the macro

```
@my_time_fixed (macro with 1 method)
```

```julia
begin
    macro my_time_fixed(ex)
        quote
            t0 = time()
            result = $(esc(ex))
            println("Elapsed: ", time() - t0)
            result
        end
    end
end
```

```julia
begin
    @my_time println(t0 + 1)
    @my_time_fixed println(t0 + 1)
end
```

```
1.766150691942e9
Elapsed: 0.0
101
Elapsed: 0.0
```

As a rule of thumb, you should always use `esc` to isolate the macro's code from the interpolated part (unless variable capture is the wanted behaviour, which can happen).

```julia
md"""As a rule of thumb, you should always use `esc` to isolate the macro's code from
the interpolated part (unless variable capture is the wanted behaviour, which can
happen)."""
```

Section 6 - Generated Functions

This is gonna get a bit more funky, but you can go a step further by generating function code at compile time, dependant of the argument types. This is done with the `@generated` macro. Step by step, it looks like this:

- Julia sees a call: f(x)
- Julia infers the types of x
- Julia sees f is @generated
- Julia calls the generator with the inferred types
- The generator returns an `Expr`
- That expression is compiled as the method body
- Runtime execution happens later

Generated functions allow you to create new functions programatically. This is similar to what you do with function parametrization using type overloading, but with a lot more control.

Demonstration:

```
begin
    function sum_tuple(t::NTuple{N,Int}) where N
        s = 0
        for i in 1:N
            s += t[i]
        end
        s
    end
    println(sum_tuple((1, 2, 3)))
end
```

```
6
```

This is simple, and it works, however it's not guaranted the compiler will unroll the loop (serializing it as a series of sum).

```
println(@code_typed sum_tuple((1, 2, 3)))
```

```
CodeInfo(
1 ─       nothing::Nothing
2 ┄ %2  = φ (#1 => 1, #6 => %13)::Int64
│   %3  = φ (#1 => 1, #6 => %14)::Int64
│   %4  = φ (#1 => 0, #6 => %7)::Int64
│   %5  = $(Expr(:boundscheck, true))::Bool
│   %6  =   builtin Base.getfield(t, %2, %5)::Int64
│   %7  = intrinsic Base.add_int(%4, %6)::Int64
│   %8  =   builtin (%3 === 3)::Bool
└──       goto #4 if not %8
3 ─       goto #5
4 ─ %11 = intrinsic Base.add_int(%3, 1)::Int64
└──       goto #5
5 ┄ %13 = φ (#4 => %11)::Int64
│   %14 = φ (#4 => %11)::Int64
│   %15 = φ (#3 => true, #4 => false)::Bool
│   %16 = intrinsic Base.not_int(%15)::Bool
└──       goto #7 if not %16
6 ─       goto #2
7 ─       return %7
) => Int64
```

Those gotos are the loop, in compiler language. Let's build a version that explicitely unrolls the loop.

```
begin
    @generated function sum_tuple_generated(t::NTuple{N,Int}) where N
    terms = [:(t[$i]) for i in 1:N]
    reduce((a,b) -> :($a + $b), terms)
    end
    println(sum_tuple_generated((1, 2, 3)))
end
```

```
6
```

We now programatically generate code that sums all the elements in the tuple (in other word, from the tupe we litterally generate `t[1] + t[2] + t[3]...`) If we inspect the code, you'll see that **there's no loop**

```julia
println(@code_typed sum_tuple_generated((1, 2, 3)))
```

```
CodeInfo(
1 ─ %1 = $(Expr(:boundscheck, true))::Bool
│   %2 =   builtin Base.getfield(t, 1, %1)::Int64
│   %3 = $(Expr(:boundscheck, true))::Bool
│   %4 =   builtin Base.getfield(t, 2, %3)::Int64
│   %5 = intrinsic Base.add_int(%2, %4)::Int64
│   %6 = $(Expr(:boundscheck, true))::Bool
│   %7 =   builtin Base.getfield(t, 3, %6)::Int64
│   %8 = intrinsic Base.add_int(%5, %7)::Int64
└──      return %8
) => Int64
```

Metaprogamming tools are useful to manipulate the code itself, however it should be used sparingly (as it is not very intuitive to read), do not use it when:

- A function works
- Multiple dispatch works
- Clarity suffers too much

Thank you for your attention :D