100 days of code with Julia

Sergio-Feliciano Mendoza-Barrera

December 3, 2020

The today's topic is metaprogramming basics.

Metaprogramming in Julia. Day 1

Julia code can be represented as a data structure of the language itself. This allows a program to transform and generate its own code (Lauwens & Downey, 2020, p. 204).

Expressions

Every Julia program starts as a string

```
prog = "1 + 2"
ex = Meta.parse(prog)
```

we can get the type of the variable with `typeof` as usual and `dump` the tree structure[1].

[1] The `dump` function displays `expr` objects with annotations.

```
typeof(ex)
dump(ex)
```

```
Expr
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

Expressions can be constructed directly by prefixing with : inside parentheses or using a `quote` block

```
ex = quote
    1+2
end
```

Now, *Julia can evaluate an expression object* using `eval`

```
eval(ex)
```

```
3
```

Every module has its own `eval` function that evaluates expressions in its scope[2].

[2] *WARNING*: When you are using a lot of calls to the function `eval`, often this means that something is wrong. `eval` is considered *evil*.

Macros. Day 2

Macros can include generated code in a program. A macro maps a tuple of `Expr` objects directly to a compiled expression:

Here is a simple macro

```
macro containervariable(container, element)
    return esc(:($(Symbol(container,element)) = $container[$element]))
end
```

Macros are called by prefixing their name with the `@` (at-sign). The macro call `@containervariable letters 1` is replaced by:

```
:(letters1 = letters[1])
```

`@macroexpand @containervariable letters 1` returns this expression which is extremely useful for debugging.

```
@macroexpand @containervariable letters 1
```

This example illustrates how a macro can access the name of its arguments, something a function can't do. The return expression needs to be *escaped* with `esc` because it has to be resolved in the macro call environment[3].

[3] *Note:* Why macros? Macros generate and include fragments of customized code during parse time, thus before the full program is run.

A concrete example

A concrete example of the macro usage is show here [4], in this case we are going to use the `@elapsed` macro over the `peakflops` function

[4] Edelman, A., Sanders, D. P., Sanderson, G., Schloss, J., & Forget, B. (2020). Introduction to computational thinking. Online. Retrieved from https://computationalthinking.mit.edu/Fall20

```
peakflops()
```

```
5.892558158665142e10
```

now we can see how long the operation took

```
@elapsed peakflops()
```

```
0.346514384
```

Internally, Julia take this piece of code and transform in another piece of code

```
@macroexpand @elapsed peakflops()
```

```
quote
    #= timing.jl:231 =#
    while false
        #= timing.jl:231 =#
    end
    #= timing.jl:232 =#
    local var"#5#t0" = Base.time_ns()
    #= timing.jl:233 =#
    peakflops()
    #= timing.jl:234 =#
    (Base.time_ns() - var"#5#t0") / 1.0e9
end
```

the comments indicate the place where Julia inserts extra code for this specific macro

```
Base.remove_linenums!(@macroexpand @elapsed peakflops())
```

```
quote
    while false
    end
    local var"#7#t0" = Base.time_ns()
    peakflops()
    (Base.time_ns() - var"#7#t0") / 1.0e9
end
```

Bibliography

Edelman, A., Sanders, D. P., Sanderson, G., Schloss, J., & Forget, B. (2020). Introduction to computational thinking. Online. Retrieved from https://computationalthinking.mit.edu/Fall20.

Lauwens, B. & Downey, A. B. (2020). *Think Julia: How to Think Like a Computer Scientist* (1st ed.). O'Reilly. Retrieved from https://benlauwens.github.io/ThinkJulia.jl/latest/book.html.