Table of Contents


```
begin
    using PlutoUI
    PlutoUI.TableOfContents(indent=true, depth=4, aside=true)
end
```

The structure of code

One somewhat unusual feature of Julia (originating from its Lisp heritage) is that it allows us to talk about Julia code – from within Julia itself!

That is, we can capture a piece of Julia code into a Julia object, which we can then inspect and modify.

Once we have the modified code, we can then evaluate it.

An easy way to create a piece of code is by parsing a string, i.e. interpreting the string as Julia code and returning a Julia object that represents that piece of code. Nonetheless, in the future we will prefer, when possilbe, to create Julia objects directly, rather than from strings, since strings are "opaque objects" that Julia does not understand.

Exercise

1. Define a variable code to be as the result of parsing the string "j = i^2" using the function `Meta.parse`.
2. What type is the object code? Note that code is just a normal `Julia` variable, of a particular special type.
3. Use the `dump` function to see what there is inside code. Remembering that code is just a particular kind of `Julia` object, use the `Julia` to play around interactively, seeing how you can extract pieces of the code object.
4. How is the operation `i^2` represented? What kind of object is that subpiece?
5. Copy code into a variable `code2`. Modify this to replace the power 2 with a power 3. Make sure that the original code variable is not also modified.
6. Copy `code2` to a variable `code3`. Replace `i` with `i + 1` in `code3`.
7. Define a variable `i` with the value 4. Evaluate the different code expressions using the `eval` function and check the value of the variable `j`.

```
code = :(j = i ^ 2)
```

```
code = Meta.parse("j = i^2")
```

Expr

```
typeof(code)
```

```
Expr
  head: Symbol =
  args: Array{Any}((2,))
    1: Symbol j
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol ^
        2: Symbol i
        3: Int64 2
```

```
with_terminal() do
    dump(code)
end
```

```
(:head, :args)
```
```
propertynames(code)
```

```
(:(i ^ 2), Expr)
```
```
code.args[2], typeof(code.args[2])
```

```
code₂ = :(j = i ^ 2)
```
```
code₂ = code |> deepcopy
```

```
3
```
```
code₂.args[2].args[3] = 3
```

```
(:(j = i ^ 2), :(j = i ^ 3))
```
```
code, code₂
```

```
code₃ = :(j = i ^ 3)
```
```
code₃ = code₂ |> deepcopy
```

```
:i
```
```
code₃.args[2].args[2] # = :(i + 1)
```

```
:(i + 1)
```
```
code₃.args[2].args[2] = :(i + 1)
```

```
:(j = (i + 1) ^ 3)
```
```
code₃
```

```
125
```
```
begin
    i = 4
    eval(code₃)  # ≡ (4+1)^3 == 125
end
```

Symbols

Let's think about what code looks like. It's made up of characters joined into words, plus certain types of punctuation, for example

```
function f()
    ...
end
```

and

```
for
    ...
end
```

and

```
z = x + y
result = first + second
```

The smallest building blocks, or atoms, of code are variable names like `x` and `first`, and other symbols like `+`. `Julia` calls all of these names symbols.

When you type `+` or `x` in the `REPL`, `Julia` immediately tries to evaluate the result: it treats `+` or `x` as a name or binding that points to an object, and it tries to return the object itself.

For metaprogramming, we instead need a way to talk just about "the unevaluated piece of code consisting of the name x". We write this using a colon, `:`, e.g. `:x` and `:+`:

```
s = :x
```
```
s = :x
```

```
Symbol
```

```
typeof(s)
```

We can think of `:hello`, for example, as a way to talk about a possible variable called `hello`, without evaluating it.

Expression

Anything more complicated than a single symbol is an expression, for example `x + y`, which means "call the function named `+` on the objects named `x` and `y`". (This is equivalent to writing `+(x, y)` in `Julia`)

We can talk about the piece of code "x + y" without evaluating it by quoting it, again using a colon:

```
:(x + y)
```

Let's give that a name. We will use `expr` for expression:

```
expr = :(x + y)
```

```
expr = :(x + y)
```

In words we are telling `Julia` something like: "Define a new variable, called `expr`, whose value is bound to the unevaluated piece of code `x + y`.

Note that if we just type `x + y` into the REPL, again Julia will immediately try to evaluate this by looking up the values that are bound to the variable names `x` and `y`. But we haven't yet defined variables called `x` and `y`, so this will error.

However, there is no such problem with talking about "the piece of code x + y" – here, `x` and `y` are just symbols.

```
try
    x + y
catch e
    println("Intercepted Error ", e)
end
```

```
Intercepted Error UndefVarError(:x)
```

The Expr type and the structure of expressions

We now have a `Julia` object with the name `expr` that is referring to an unevaluated piece of `Julia` code, `x + y`. Let's use `Julia`'s great introspection tools to look inside this object and see how it's formed!

Firstly let's look at its type:

```
Expr
```

```
typeof(expr)
```

`Expr` is a type representing any `Julia` expression that is more complicated than a single symbol.

How is the sum represented? Let's find out using `dump`:

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

```
with_terminal() do
    dump(expr)
end
```

Alternatively we can use `expr<TAB>` interactively, or `propertynames(expr)` programatically, to expression object's attributes:

```
(:head, :args)
```

```
propertynames(expr)
```

We see that this expression (and, in fact, any expression!) consists of two pieces:

- a head and
- a collection of `args`, i.e. arguments.

`:call`

```
expr.head
```

In this case, the head is `:call`, which tells us that this piece of code is a function call.

```
[:+, :x, :y]
```

```
expr.args
```

The arguments show us the pieces of the function call: the function to be called corresponds to the symbol `:+`, and that function is called on the objects corresponding to the symbols `:x` and `:y`.

Note that the type of args is a vector of `Any`. This is because we can have things other than symbols as an argument, e.g.

```
:(x + 1) |> dump
```

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

Creating code from scratch

Since `Expr` is just a standard `Julia` type, we can create objects of that type in the standard way, namely by calling the constructor function of the type. The constructor for `Expr` just accepts a tuple of its arguments, starting with head and following with a list of arguments, e.g.

`:(x + y)`

```
Expr(:call, :+, :x, :y)
```

More complicated expressions

Now let's look at an expression that's a bit more complicated, with two operations:

```
nexpr = :(x + y * z)
```

```
nexpr = :(x + y * z)
```

How can this expression be represented?

Firstly `Julia` needs to decide which operation will be done first: does this mean `x + (y * z)`, or `(x + y) * z`. This choice is made by having a table of operator precedence; in this particular case, `*` has a higher precedence than `+`, so will be "bound more tightly", or done first, so the expression will be interpreted as `x + (y * z)`. Operator precedence is fixed in the parser and cannot be changed (without modifying and recompiling `Julia`). If you need a different order of operations, use parentheses!

So how does `Julia` represent this expression? It will be a sum of two things, `x` and `y * z`. What is `y * z`? Well, we already know that it's... an expression, an object of type `Expr`!

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol *
    2: Symbol y
    3: Symbol z
```

```
with_terminal() do
    :(y * z) |> dump
end
```

It has the same overall structure as `:(x + y)`, but the arguments are different.

Exercise

Can you now guess how `:(x + (y * z))` is represented? Try to predict the answer before moving on.

```
## Expr(:call :+ x (:call :* y z))
```

```
:(x + (y * z)) |> dump
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Symbol x
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol *
        2: Symbol y
        3: Symbol z
```