Manipulating expression trees

Table of Contents


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

Structure of complex expressions

Recall from the previous notebook that we want to know how `Julia` represents a complex expression with more than one operation, e.g.:

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

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

```
expr |> 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
```

We see that indeed we have one `Expr` object embedded into another! That is, one of the args of the top-level expression `expr` is another object of type `Expr`:

```
(:(y * z), Expr)
```

```
expr.args[3], typeof(expr.args[3])
```

We can say that the `Expr` type in `Julia` is a recursive data type: an object of type `Expr` can contain other `Expr` objects! This is another good reason why the type of the args field is a vector of `Any`!

This fact has important implications for how we need to work with `Expr` objects.

Another example is an assignment:

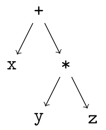
```
:(x = y + z) |> dump
```

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

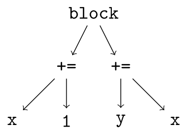
As we have seen, any piece of code that is more complicated than a single function call has a hierarchical, or nested structure, with Exprs contained inside other Exprs. In this section we will see how to manipulate such structures. It is common to think of them as trees, called abstract syntax trees or ASTs, or computational graphs.

There are several packages that enable us to visualise an expression as a tree, for example the `TreeView.jl` package that I wrote, and `GraphRecipes.jl`.

```julia
using TreeView
```




```julia
@tree x + y * z
```




```julia
begin
    expr₁ = quote
        x += 1
        y += x
    end

    expr₁ = Base.remove_linenums!(expr₁)

    TreeView.walk_tree(expr₁)
end
```

Walking through expression trees

Since complex expressions have a recursive structure, we need to take this into account when we process them. As a simple example, let's try to solve the following problems:

```
Given an expression, substitute y for x, i.e. replace each x in the expression with a y.
```

The easiest way to do this is to modify the expression.

How can we do this? We will need to check each argument to see if it's an `:x`. If it is, we replace it; if not, we move on:

Exercise

- Write a function `substitute!` that takes an expression and replaces each `:x` with `:y`.
- Test it on the expression `x + x * y`. Does it work correctly? If not, what is it missing?

```
substitute! (generic function with 3 methods)
```

```julia
function substitute!(expr::Expr; repl_rule=:x => :z)
    args = expr.args

    for ix ∈ 1:length(args)
        if args[ix] == :x
            args[ix] = :y # hard coded replace
        elseif args[ix] isa Expr
            substitute!(args[ix])
        elseif args[ix] isa Symbol
            substitute!(args[ix],repl_rule)
        else
            throw(
                ArgumentError("$(args[ix]) of type $(typeof(args[ix])) not yet supported")
            )
        end
    end
    expr
end
```

```
substitute! (generic function with 3 methods)
```

```julia
substitute!(expr::Symbol, repl_rule=:x => :z) = expr == repl_rule[1] ? repl_rule[2] : expr
```

3 methods for generic function **substitute!**:

- substitute!(expr::**Symbol**) in Main.var"workspace#63" at /home/pascal/Projects/ML_DL/Notebooks/julia-notebooks/JuliaCon2021/Metaprogramming/3 - Expression trees.jl#==#b77c82e7-3836-4e94-add1-87b79c770426:1
- substitute!(expr::**Symbol**, repl_rule) in Main.var"workspace#63" at /home/pascal/Projects/ML_DL/Notebooks/julia-notebooks/JuliaCon2021/Metaprogramming/3 - Expression trees.jl#==#b77c82e7-3836-4e94-add1-87b79c770426:1
- substitute!(expr::**Expr**; *repl_rule*) in Main.var"workspace#150" at /home/pascal/Projects/ML_DL/Notebooks/julia-notebooks/JuliaCon2021/Metaprogramming/3 - Expression trees.jl#==#2a778bd6-7f05-4c29-bacd-322370cf84d2:1

```julia
methods(substitute!)
```

```
(:z, :y)
```

```julia
substitute!(:x), substitute!(:y)
```

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

```julia
substitute!(:(x + y))
```

```julia
try
    substitute!(:(x + :x - y))
catch err
    println("Intercpeted error: $(err)")
end
```

```
Intercpeted error: ArgumentError(":x of type QuoteNode not yet supported")
```

```
expr₂ = :(x + x * y)
```

```julia
expr₂ = :(x + x * y)
```

```
:(y + y * y)
```

```julia
substitute!(expr₂)
```

Now all x s were successfully replaced! But note that the original expression has been lost:

```
:(y + y * y)
```

```julia
expr₂
```

Although the code is easier to write using mutation (since otherwise we need to build up a similar expression with the same structure, which is possible but trickier), users probably don't expect or want their expression to be mutated, since you cannot then recover the original expression, which you might need later. It is common to then provide a non-mutating version, by making a copy of the original expression and mutating that.

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

```julia
substitute(expr) = deepcopy(expr) |> substitute!
```

```
expr₃ = :(x + x * y)
```

```julia
expr₃ = :(x + x * y)
```

```
(:(y + y * y), :(x + x * y))
```

```julia
substitute(expr₃), expr₃
```

Note that it would have been tempting to write this by iterating over the arguments using `for arg ∈ expr.args ...`, but that will not work, since you then cannot modify the resulting argument in place within the expression – arg is a copy of the immutable object inside the expression, rather than a reference to it:

```
expr₄ = :(x + y)
```

```julia
expr₄ = :(x + y)
```

```
for arg ∈ expr₄.args
    @show arg, typeof(arg)
    arg = :z
end
```

```
(arg, typeof(arg)) = (:+, Symbol)
(arg, typeof(arg)) = (:x, Symbol)
(arg, typeof(arg)) = (:y, Symbol)
```

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

```
expr₄
```

Exercise

- Make the substitute function more general, by specifying what to substitute with what

```
## done above
```

Exercise

- Find which variables are used inside an expression
- For example, the expression `2x + y * x - 1` should return the vector `[x, y]`, removing duplicates. *Hint*: This requires returning the vector of variables.

find_vars

find_vars find the variable inside the expression variable are supposed to be only alphabetical chars upcase/lowcase

```
"""
  find_vars find the variable inside the expression
  variable are supposed to be only alphabetical chars upcase/lowcase
"""
function find_vars(expr)
    s = Set{Any}()
    s = _find_vars(expr, s)
    Vector{Any}(collect(s)) |> sort
end
```

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

```
_find_vars(expr::Number, s::Set)::Set = s
```

```
_find_vars (generic function with 2 methods)
```

```
_find_vars(expr::Symbol, s::Set)::Set = match(r"\A[a-zA-Z]+\z", string(expr)) !== nothing ? push!(s, expr) : s
```

```
_find_vars (generic function with 3 methods)
```

```
function _find_vars(expr::Expr, s::Set)::Set
    for arg ∈ expr.args
        _find_vars(arg, s)
    end
    s
end
```

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

```
find_vars(:(2x + y * x - 1))
```

```
[:y]
```

```
find_vars(:(2 + y))
```

```
[:x, :y, :z]
```

```
find_vars(:((2 + y) * (x - 3) / (z - x)))
```

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

```
find_vars(:(x + x * y))
```

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

```
find_vars(:(2x * y))
```

Remark But first let's remark that it is usually an anti-pattern to use `isa` in `Julia`! [Recall that `x isa T` tests if the object `x` has type `T`.]

This should usually be replaced by `dispatch!`. We can rewrite the same code as follows:

```
find_variables (generic function with 3 methods)
```

```
begin
    find_variables(x::Symbol) = [x]

    find_variables(x::Real) = []

    find_variables(x::Expr) = sort(unique(reduce(vcat, find_variables(arg) for arg ∈ x.args[2:end])))
end
```

Quote blocks

For longer pieces of code it is common to use `quote ... end` instead of `:( ... )`:

```
expr₅ =
quote
    #= /home/pascal/Projects/ML_DL/Notebooks/julia-notebooks/JuliaCon2021/Metaprogramming/3 - Expression trees.jl#==#85aac
    x += 1
    #= /home/pascal/Projects/ML_DL/Notebooks/julia-notebooks/JuliaCon2021/Metaprogramming/3 - Expression trees.jl#==#85aac
    y += x
end
```

```
expr₅ = quote
    x += 1
    y += x
end
```

Note that quote blocks automatically embed line number information telling you where that piece of code was created, for debugging purposes. We can remove that with `Base.remove`

```
quote
    x += 1
    y += x
end
```

```
Base.remove_linenums!(expr₅)
```

```
Base.remove_linenums!(expr₅) |> dump
```

```
Expr
  head: Symbol block
  args: Array{Any}((2,))
    1: Expr
      head: Symbol +=
      args: Array{Any}((2,))
        1: Symbol x
        2: Int64 1
    2: Expr
      head: Symbol +=
      args: Array{Any}((2,))
        1: Symbol y
        2: Symbol x
```

The only new feature here is that a quote block is a new type of `Expr` with head equal to `:block`.

Further exercises

- Given an expression, wrap all the literal values (numbers) in the expression with a wrapper type.
- Replace each of `+`, `-`, `*` and `/` with the corresponding checked operation, which checks for overflow. E.g. + should be replaced by `Base.checked_add`.