JuMP's macros: why are they needed?

Oscar Dowson

JuMP-dev 2026

JuMP's macros

The basic syntax

```
using JuMP
model = Model()
@variable(model, x[1:10] >= 0, Int)
@expression(model, expr, sum(i^1.5 * x[i] for i in 1:n))
@objective(model, Min, 3.0 * expr + x[1])
@constraint(model, c[i in 1:10], x[i] <= i)
```

The purpose of this talk

Why is this talk needed:

- JuMP's macros are hard to understand
- We get frequent questions about them
- Why do we even have them?

By the end of this talk you will:

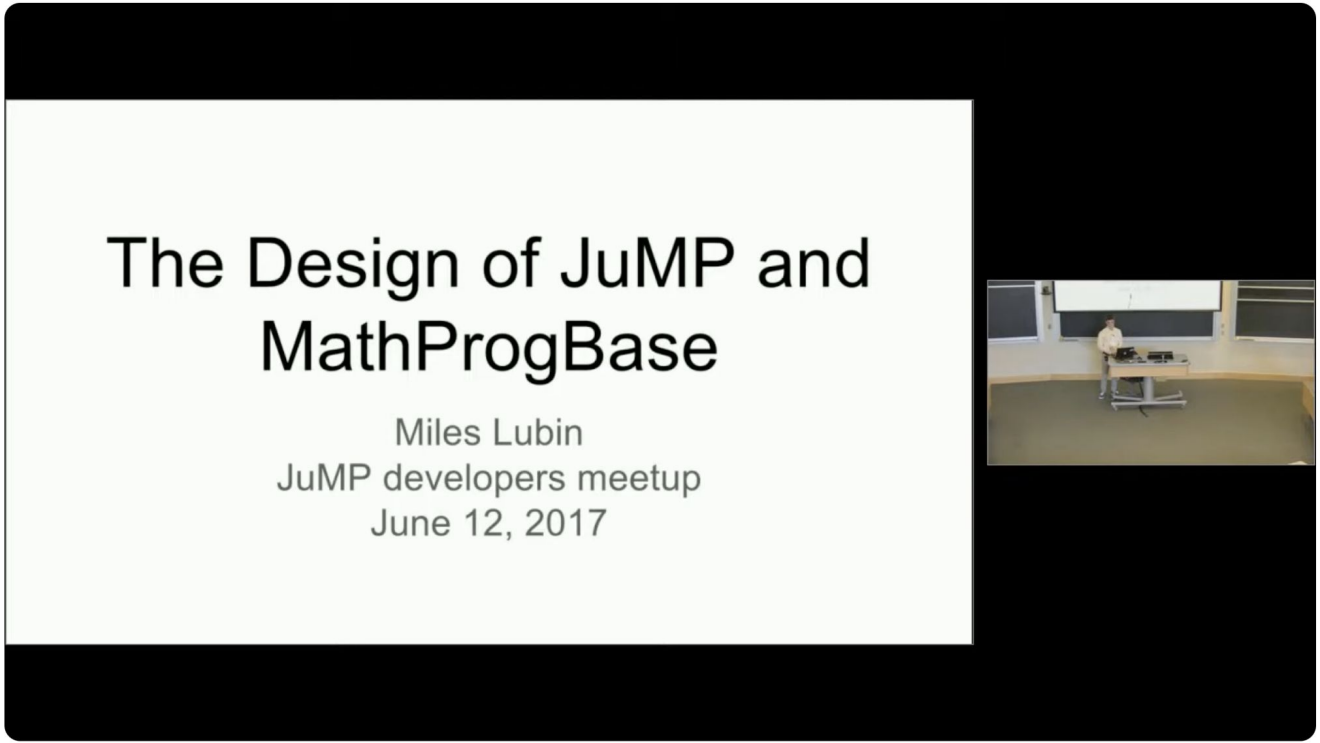
1. Understand the performance trade-offs of data structures for linear expressions
2. Know what a Julia macro is
3. Know why we use them in JuMP
4. Know how and when to use them in your code

By the end of this talk you will not:

- Be able to write your own macros

The 2017 JuMP-dev workshop

The last time we talked about JuMP's macros



The design of JuMP and MathProgBase (Miles Lubin, MIT)

Representing linear expressions

A basic data structure

```
struct Variable
   index::Int
end

mutable struct AffExpr
   terms::Dict{Variable,Float64}
   constant::Float64
end

function add_mul!(
    f::AffExpr,
    c::Float64,
    x::Variable,
)
    old_c = get(f.terms, x, 0.0)
    f.terms[x] = old_c + c
end
```

How can we efficiently construct this expression?

```
sum(i^1.5 * Variable(i) for i in 1:n)
```

Manual construction

Efficient but not readable

```
# y = sum(i^1.5 * Variable(i) for i in 1:n)

y = zero(AffExpr)
for i in 1:n
    add_mul!(y, i^1.5, Variable(i))
end
```

Operator overloading

Multiple dispatch makes this easy

```
Base.zero(::Type{AffExpr}) =
    AffExpr(Dict{Variable,Float64}(), 0.0)

Base.:*(c::Float64, v::Variable) =
    AffExpr(Dict(v => c), 0.0)

function Base.:+(x::AffExpr, y::AffExpr)
    z = AffExpr(copy(x.terms), x.constant)
    z.constant += y.constant
    for (x_i, c_i) in y.terms
        add_mul!(z, c_i, x_i)
    end
    return z
end
```

```
y = sum(i^1.5 * Variable(i) for i in 1:n)

# becomes

y1 = *(1^1.5, Variable(1))
y2 = *(2^1.5, Variable(2))
y3 = +(y1, y2)
y4 = *(3^1.5, Variable(3))
y5 = +(y3, y4)
y6 = *(4^1.5, Variable(4))
y7 = +(y5, y6)
y8 = ...
```

Expression trees

Pyomo uses these

```julia
@enum(Class, kConst, kVar, kMul, kAdd)
struct Node
   class::Class
   children::Vector{Node}
   data::Float64
end
Node(x::Variable) = Node(kVar, Node[], x.index)
Node(x::Float64) = Node(kConst, Node[], x)
Base.:*(c::Float64, v::Variable) = Node(c) * Node(v)
Base.:*(x::Node, y::Node) = Node(kMul, [x, y], 0.0)
Base.:+(x::Node, y::Node) = Node(kAdd, [x, y], 0.0)
function flatten(g::Node)
   f = zero(AffExpr)
   stack = Any[g]
   while !isempty(stack)
       arg = pop!(stack)
       # ... lines omitted ...
   end
   return f
end
```

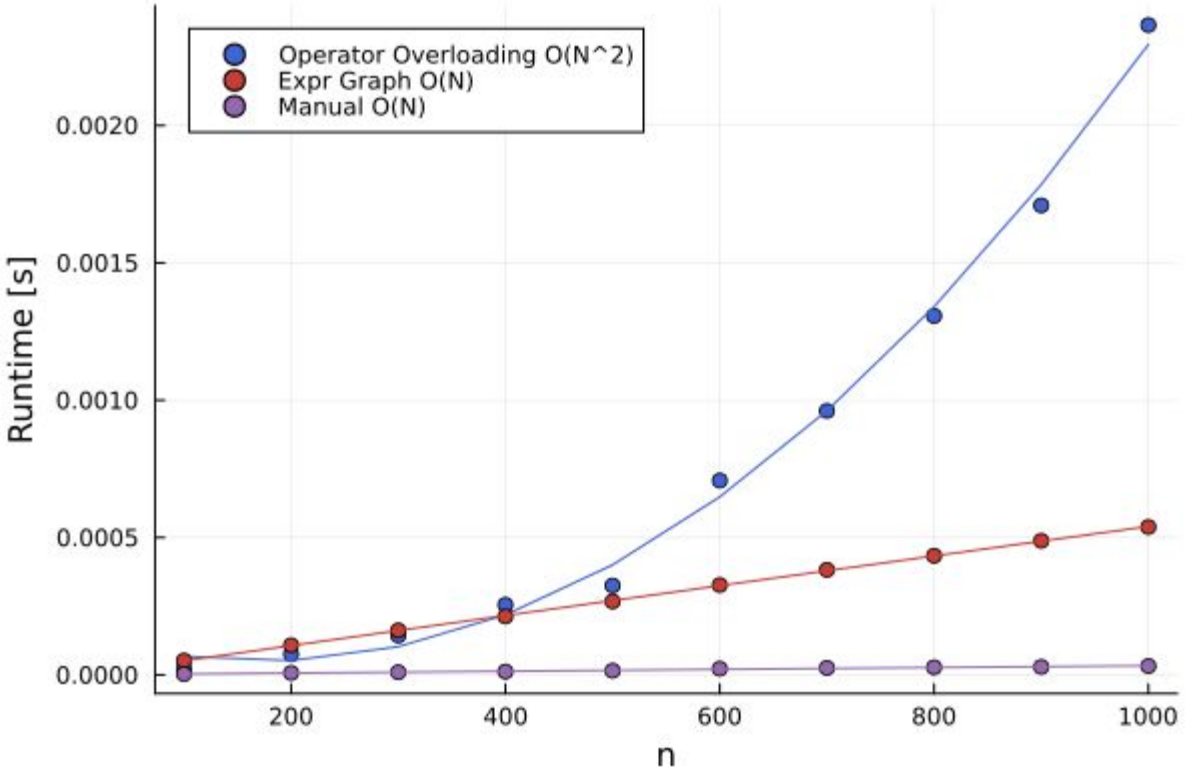
```julia
y = sum(i^1.5 * Variable(i) for i in 1:n)

# becomes

y_graph = Node(kAdd, [
  Node(kMul, [
    Node(kConst, [], 1^1.5),
    Node(kVar, [], 1.0),
    ], 0.0,
  ),
  Node(kMul, [
    Node(kConst, [], 2^1.5),
    Node(kVar, [], 2.0),
    ], 0.0,
  )], 0.0)
)
y = flatten(y_graph)
```

Benchmarking performance

Three different approaches



Representing linear expressions

A summary of the tradeoffs

Approach	Syntax	Speed	Readability
Operator overloading	`y = sum(i^1.5 * Variable(i) for i in 1:n)`	Slow	Good
Expression graph	`y = sum(i^1.5 * Variable(i) for i in 1:n)` `y = flatten(y)`	Okay	Okay

Manual construction	`y = zero(AffExpr)`
	`for i in 1:n`
	`  add_mul!(y, i^1.5, Variable(i))`
	`end`
		Fast	Poor

gurobipy.quicksum

One approach

```
quicksum(data)
```

A version of the Python `sum` function that is much more efficient for building large Gurobi expressions (`LinExpr` or `QuadExpr` objects). The function takes a list of terms as its argument.

Note that while `quicksum` is much faster than `sum`, it isn't the fastest approach for building a large expression. Use `addTerms` or the `LinExpr()` constructor if you want the quickest possible expression construction.

PARAMETERS:

data – List of terms to add. The terms can be constants, `Var` objects, `LinExpr` objects, or `QuadExpr` objects.

RETURNS:

An expression that represents the sum of the terms in the input list.

EXAMPLE:

```
expr = gp.quicksum([2*x, 3*y+1, 4*z*z])
expr = gp.quicksum(model.getVars())
```

What is a macro?

A macro is a “function” that rewrites code

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

julia> dump(:(x - y))
Expr
 head: Symbol call
 args: Array{Any}((3,))
   1: Symbol -
   2: Symbol x
   3: Symbol y
```

```
julia> macro replace_plus(expr::Expr)
         if Meta.isexpr(expr, :call, 3) && expr.args[1] == :+
           return esc(:($(expr.args[2]) - $(expr.args[3])))
         end
         return expr
       end
@replace_plus (macro with 1 method)

julia> @replace_plus 10 + 3
7

julia> @replace_plus 10 * 3
30

julia> @macroexpand @replace_plus foo + bar
:(foo - bar)
```

A @rewrite macro

The simplest case

```
julia> macro rewrite(expr)
          body, loop = expr.args[2].args[1], expr.args[2].args[2]
          return quote
              y = zero(AffExpr)
              for $(loop.args[1]) in $(loop.args[2])
                  add_mul!(y, $(body.args[2]), $(body.args[3]))
              end
              y
          end |> esc
      end

julia> @macroexpand @rewrite sum(i^1.5*Variable(i) for i in 1:n)
quote
   y = zero(AffExpr)
   for i = 1:n
       add_mul!(y, i ^ 1.5, Variable(i))
   end
   y
end
```

MutableArithmetics.jl

The proper version of @rewrite

```
julia> import MutableArithmetics as MA

julia> @macroexpand(
         MA.@rewrite(sum(i^1.5*Variable(i) for i in 1:n), move_factors_into_sums = false)
       )
quote
    var"#111###300" = MA.Zero()
    for i = 1:n
        var"#109###301" = i ^ 1.5
        var"#110###302" = Variable(i)
        var"#111###300" =
            MA.operate!!(MA.add_mul, var"#111###300", var"#109###301", var"#110###302")
    end
    var"#111###300"
end
```

@expression

“Syntactic sugar for MutableArithmetics”

```
julia> @macroexpand @expression(model, sum(i^1.5 * x[i] for i in 1:n))
quote
    JuMP._valid_model(model, :model)
    var"#10#start_time" = JuMP.time()
    var"#14###280" = MA.Zero()
    for i = 1:n
        var"#13###281" = i ^ 1.5
        var"#14###280" = MA.operate!!(MA.add_mul, var"#14###280", var"#13###281", x[i])
    end
    var"#15###282" = flatten!(var"#12###280")
    var"#11###283" = JuMP._replace_zero(model, var"#15###282")
    JuMP._add_or_set_macro_time(model, <OMITTED>, JuMP.time() - var"#10#start_time")
    var"#11###283"
end
```

@expression

“Syntactic sugar for MutableArithmetics” + containers

```
julia> @macroexpand @expression(model, expr[i in 1:3], x[i])
quote
    JuMP._valid_model(model, :model)
    var"#38#start_time" = JuMP.time()
    JuMP._error_if_cannot_register(model, :expr)
    var"#39###291" = JuMP.Containers.container(
        (i,) -> begin
            var"#41###289" = x[i]
            var"#42###290" = flatten!(MA.copy_if_mutable(var"#41###289"))
            JuMP._replace_zero(model, var"#42###290")
        end,
        JuMP.Containers.vectorized_product(Base.OneTo(1:3)),
        JuMP.Containers.AutoContainerType,
        Any[:i],
    )
    expr = (model[:expr] = var"#39###291")
    JuMP._add_or_set_macro_time(model, (...), JuMP.time() - var"#38#start_time")
    var"#39###291"
end
```

When to use @expression

Because sometimes it isn't needed

```
julia> model = Model();

julia> @variable(model, x[1:10]);

julia> @which sum(x)
sum(a::AbstractArray{T}; dims, init) where T<:MutableArithmetics.AbstractMutable
    @ MutableArithmetics
~/.julia/packages/MutableArithmetics/2Djmb/src/dispatch.jl:33

julia> @macroexpand @expression(model, sum(x))
quote
    JuMP._valid_model(model, :model)
    var"#84#start_time" = JuMP.time()
    var"#86###306" = sum(x)
    var"#88###308" = flatten!(MA.copy_if_mutable(var"#86###306"))
    JuMP._replace_zero(model, var"#88###308")
    JuMP._add_or_set_macro_time(model, (...), JuMP.time() - var"#84#start_time")
    var"#85###309"
end
```

When to use @expression

Because sometimes it isn't needed

You need @expression

- If you use the sum(expr for index in set) syntax
- If you use the basic arithmetic operators: +, -, *
- If you are using the container syntax
- For consistency

You don't need @expression

- For linear algebra
- If performance doesn't matter

When in doubt, benchmark. Or use @macroexpand to see what code is being run.

@objective

Rewrite the function and set the objective sense

```
julia> @macroexpand @objective(model, Max, sum(i^1.5 * x[i] for i in 1:3))
quote
    JuMP._valid_model(model, :model)
    var"#27#start_time" = JuMP.time()
    var"#31###285" = MA.Zero()
    for i = 1:3
        var"#30###286" = i ^ 1.5
        var"#31###285" = MA.operate!!(MA.add_mul, var"#31###285", var"#30###286", x[i])
    end
    var"#32###287" = flatten!(var"#29###285")
    var"#32###287" = JuMP._replace_zero(model, var"#32###287")
    JuMP.set_objective(model, MAX_SENSE, var"#32###287")
    JuMP._add_or_set_macro_time(model, <OMITTED>, JuMP.time() - var"#27#start_time")
    var"#28###287"
end
```

@constraint

A lot can happen here; this is the simplest case

```
julia> @macroexpand @constraint(model, sum(x) <= 1)
quote
    JuMP._valid_model(model, :model)
    var"#71#start_time" = JuMP.time()
    var"#74###300" = sum(x)
    var"#75###301" = MA.copy_if_mutable(var"#74###300")
    var"#76###302" = MA.operate!!(MA.sub_mul, var"#75###301", 1)
    var"#77###303" = flatten!(var"#76###302")
    var"#78#build" = JuMP.model_convert(
        model,
        JuMP.build_constraint(
            error, JuMP._functionize(var"#77###303"), LessThanZero(),
        ),
    )
    var"#72###304" = JuMP.add_constraint(model, var"#78#build", "")
    JuMP._add_or_set_macro_time(model, <OMITTED>, JuMP.time() -
var"#71#start_time")
    var"#72###304"
end
```

@variable

A lot can happen here; this is the simplest case

```
julia> @macroexpand @variable(model, x >= 0, Int)
quote
    JuMP._valid_model(model, :model)
    var"#10#start_time" = JuMP.time()
    JuMP._error_if_cannot_register(model, :x)
    var"#12#info" =
        JuMP.VariableInfo(true, 0, false, NaN, false, NaN, false, NaN, false, true)
    var"#12#variable" = JuMP.build_variable(error, var"#12#info")
    var"#12#variable" = JuMP.model_convert(model, var"#12#variable")
    var"#11###278" = JuMP.add_variable(
        model,
        var"#12#variable",
        JuMP.set_string_names_on_creation(model) ? "x" : "",
    )
    x = (model[:x] = var"#11###278")
    JuMP._add_or_set_macro_time(model, (<OMITTED>), JuMP.time() - var"#10#start_time")
    var"#11###278"
end
```

Other complications

The expanded macro code is messier than I've shown

```
julia> @macroexpand @expression(model, sum(i^1.5 * x[i] for i in 1:3))
quote
    JuMP._valid_model(model, :model)
    var"#10#start_time" = JuMP.time()
    var"#14###280" = MA.Zero()
    for i = 1:n
        var"#13###281" = i ^ 1.5
        var"#14###280" = MA.operate!!(MA.add_mul, var"#14###280", var"#13###281", x[i])
    end
    var"#15###282" = flatten!(var"#12###280")
    var"#11###283" = JuMP._replace_zero(model, var"#15###282")
    JuMP._add_or_set_macro_time(model, <OMITTED>, JuMP.time() - var"#10#start_time")
    var"#11###283"
end
```

Other complications

The expanded macro code is messier than I've shown

```
julia> @macroexpand @expression(model, sum(i^1.5 * x[i] for i in 1:3))
quote
    #= REPL[10]:1 =#
    JuMP._valid_model(model, :model)
    begin
        #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:418 =#
        var"#89#start_time" = JuMP.time()
        #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:419 =#
        begin
            #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:401 =#
            var"#90###309" = let model = model
                    #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:402 =#
                    begin
                        #= /Users/odow/git/jump-dev/JuMP/src/macros/@expression.jl:86 =#
                        begin
                            #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:264 =#
                            var"#91###305" = begin
                                    #= /Users/odow/.julia/packages/MutableArithmetics/2Djmb/src/rewrite.jl:387 =#
                                    let
                                        #= /Users/odow/.julia/packages/MutableArithmetics/2Djmb/src/rewrite.jl:388 =#
                                        begin
                                            #= /Users/odow/.julia/packages/MutableArithmetics/2Djmb/src/rewrite.jl:384 =#
                                            var"#93###306" = MutableArithmetics.Zero()
                                            for i = 1:3
                                                #= /Users/odow/.julia/packages/MutableArithmetics/2Djmb/src/rewrite_generic.jl:317 =#
                                                var"#92###307" = i ^ 1.5
                                                var"#93###306" = (MutableArithmetics.operate!!)(MutableArithmetics.add_mul, var"#93###306", var"#92###307", x[i])
                                            end
                                        end
                                        #= /Users/odow/.julia/packages/MutableArithmetics/2Djmb/src/rewrite.jl:389 =#
                                        var"#93###306"
                                    end
                                end
                            #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:265 =#
                            var"#94###308" = (flatten!)(var"#91###305")
                        end
                        #= /Users/odow/git/jump-dev/JuMP/src/macros/@expression.jl:89 =#
                        JuMP._replace_zero(model, var"#94###308")
                    end
                end
        end
        #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:420 =#
        JuMP._add_or_set_macro_time(model, ($(QuoteNode(:(#= REPL[10]:1 =#))), "`@expression(model, sum((i ^ 1.5 * x[i] for i = 1:3)))`"), JuMP.time() - var"#89#start_time")
        #= /Users/odow/git/jump-dev/JuMP/src/macros.jl:425 =#
        var"#90###309"
    end
end
```

Other complications

Because it's never that simple

Some things aren't linear

JuMP has three types of expression:

- `AffExpr`
- `QuadExpr`
- `NonlinearExpr`

`AffExpr` and `QuadExpr` are mutable and rewrite to mutable arithmetics. `NonlinearExpr` does not.

Maintaining MutableArithmetics.jl is hard

- There are many method overloads.
- LinearAlgebra and SparseArrays change between Julia releases
- We need to macro rewrites to ensure behavior is the same:
  - `sum(expr; init = 0)`
  - `sum(expr, init = 0)`

The purpose of this talk

Why is this talk needed:

- JuMP's macros are hard to understand
- We get frequent questions about them
- Why do we even have them?

By the end of this talk you will:

1. Understand the performance trade-offs of data structures for linear expressions
2. Know what a Julia macro is
3. Know why we use them in JuMP
4. Know how and when to use them in your code

By the end of this talk you will not:

- Be able to write your own macros