Metaprogramming

Expr, quoting, macros, hygiene, generated functions

Julia is homoiconic: code is data, represented as ordinary `Expr` objects you can build, inspect and transform, and macros run at parse time to rewrite that data before compilation. This is Lisp's model with an infix syntax, and it's used far more routinely than in most languages — `@test`, `@view`, `@.`, `@time`, `@kwdef`, `@enum`, `@threads`, `@printf` are all macros, and you'll read macro-heavy packages long before you write one.

The pipeline

```
source text
  -> parse   -> Expr (syntax tree)         <- macros operate HERE
  -> lower   -> lowered IR (SSA-ish)
  -> infer   -> typed IR                   <- @generated functions operate HERE
  -> codegen -> LLVM IR -> machine code
```

Inspect each stage:

```
Meta.parse("1 + x*2")
@macroexpand @time f(x)
@code_lowered f(1)
@code_typed f(1)
@code_warntype f(1)
@code_llvm f(1)
@code_native f(1)
```

Expr

```
ex = :(1 + x * 2)
typeof(ex)          # Expr
ex.head             # :call
ex.args             # Any[:+, 1, :(x * 2)]
dump(ex)            # full recursive structure — the tool you'll use most
Meta.@dump 1 + x*2  # same, as a macro
```

The four kinds of leaf you'll encounter:

Thing	Example	Notes
`Symbol`	`:x`, `:+`	an identifier
Literal	`1`, `"s"`, `3.0`	stored as itself
`Expr`	`:(f(x))`	a node: `head` plus `args`
`QuoteNode`	`:(:x)`	a *quoted symbol* as data, not an identifier

The `QuoteNode` distinction bites early: `:x` inside a generated expression means "the variable x", while `QuoteNode(:x)` means "the symbol `:x` as a value". `dump(:(a.b))` shows `a` as a Symbol and `b` wrapped in a `QuoteNode`.

Common heads:

```
:(f(x, y))      # head :call,      args [:f, :x, :y]
:(x = 1)        # head :(=),       args [:x, 1]
:(x + y)        # head :call,      args [:+, :x, :y]     — operators are calls
:(a.b)          # head :.,         args [:a, QuoteNode(:b)]
:(a[i])         # head :ref
:(if c; a; end) # head :if
:(for i=1:3;end)# head :for
:(function f(x); x; end)   # head :function
:(x::Int)       # head :(::)
:(begin; a; b; end)        # head :block
:(@m x)         # head :macrocall, args [Symbol("@m"), LineNumberNode, :x]
```

Building expressions by hand:

```
Expr(:call, :+, 1, 2)          # equivalent to :(1 + 2)
eval(Expr(:call, :+, 1, 2))    # 3
```

Quoting and interpolation

```
:(x + 1)                  # short quote, one expression
quote                     # long quote, a block; inserts LineNumberNodes
    a = 1
    a + 1
end

x = 5
:( $x + 1 )               # interpolate a VALUE:  :(5 + 1)
:( $(x + 1) )             # evaluate then splice: :(6)
sym = :y
:( $sym + 1 )             # :(y + 1)

args = [:a, :b, :c]
:( f($(args...)) )        # splatting into a call: :(f(a, b, c))
```

Interpolation in a quote is the mirror image of interpolation in a string, and behaves the same way. `$` reaches out into the surrounding *runtime* scope; everything else stays as syntax.

eval

```
eval(:(x = 5))            # defines x in the current module
@eval begin
    const A = 1
end
```

`eval` always operates at **module top level**, not in the local scope where you called it. You cannot `eval` a local variable into existence inside a function. Treat `eval` as a code-generation tool used at load time, not a runtime facility.

The legitimate everyday use is generating many similar definitions:

```
for op in (:+, :-, :*)
    @eval Base.$op(a::Money, b::Money) = Money($op(a.cents, b.cents))
end

for (name, unit) in [(:meters, 1.0), (:feet, 0.3048)]
    @eval $(Symbol("to_", name))(x) = x / $unit
end
# defines to_meters, to_feet
```

This pattern — a loop over names, `@eval`, `Symbol(...)` to build identifiers, `$` to splice — is 90% of the metaprogramming in real packages.

Macros

A macro receives **unevaluated syntax** and returns syntax, at parse time.

```
macro sayhello(name)
    return :( println("hello, ", $name) )
end

@sayhello "world"
@sayhello("world")        # same thing; both forms are legal
```

Inspect what it produces — always do this while developing:

```
@macroexpand @sayhello "world"
@macroexpand1 @outer @inner x    # expand only one level
```

Arguments arrive as `Expr`/`Symbol`/literals, so you can inspect and rewrite them:

```
macro showexpr(ex)
    quote
        println($(string(ex)), " = ", $(esc(ex)))
    end
end

x = 3
@showexpr x + 1          # prints "x + 1 = 4"
```

That's essentially how `@show` and `@assert` work: they stringify the syntax *and* evaluate it, which no function can do.

Hygiene and esc

Macros are hygienic: identifiers introduced inside the macro body are renamed to unique gensyms, so they can't collide with the caller's variables. This is good, and it also means that anything you want to refer to the *caller's* scope must be escaped.

```
macro double_bad(x)
    :( $x + $x )         # $x is inserted as syntax; usually fine for expressions
end

macro setlocal_bad()
    :( y = 1 )           # defines a gensym'd variable; caller's `y` is untouched
end

macro setlocal(name)
    :( $(esc(name)) = 1 )   # escaped: assigns the caller's variable
end
```

Rules of thumb:

- `esc(ex)` means "interpret this in the caller's scope".
- Escape user-supplied expressions when they must see caller-local variables. In practice, escape almost every argument.
- Do *not* escape names you introduce yourself (temporaries), or you'll clobber the caller.
- If you need a temporary in an escaped context, make it explicitly unique with `gensym()`.

```
macro twice(ex)
    tmp = gensym(:tmp)
    quote
        $(esc(ex))
        $(esc(ex))
    end
end

macro once(ex)
    tmp = gensym()
    quote
        $tmp = $(esc(ex))     # evaluate ONCE, then use twice
        ($tmp, $tmp)
    end
end
```

The double-evaluation bug is the classic macro mistake: `@twice f()` calls `f` twice, which is fine if intended and a disaster if not. Bind to a gensym first.

A useful complete example

A macro that times an expression and labels it with its own source text:

```
macro timeit(label, ex)
    quote
        local t0 = time_ns()
        local val = $(esc(ex))
```

```
        local dt = (time_ns() - t0) / 1e9
        @info "timing" label = $(esc(label)) seconds = dt
        val
    end
end

@timeit "load" load_data(path)
```

Note `local` on the macro's own temporaries (belt and braces alongside hygiene), `esc` on both user arguments, and returning `val` so the macro is expression-like rather than statement-like.

Macros that define things

```
macro defgetter(T, field)
    fname = esc(Symbol("get_", field))
    quote
        $fname(x::$(esc(T))) = getfield(x, $(QuoteNode(field)))
    end
end

@defgetter Point x     # defines get_x(p::Point) = p.x
```

Note `QuoteNode(field)` — `field` is the symbol `:x`, and we need it to appear in the generated code as the *value* `:x`, not as the identifier `x`.

Macro hygiene escape hatches

```
macro m()
    :( $(esc(:x)) )        # explicitly refer to the caller's `x` by name
end
__module__                 # inside a macro: the module it's expanded in
__source__                 # LineNumberNode of the call site
```

`__module__` is how a macro can `eval` into the caller's module or look up bindings there.

String and command macros

```
r"\d+"                     # calls @r_str("\\d+")
raw"c:\path"               # @raw_str
big"1.1"                   # @big_str
`ls -la`                   # Cmd literal, not a macro but the same family
```

Define your own:

```
macro sql_str(s)
    :( prepare_query($s) )
end

sql"SELECT * FROM t WHERE id = 1"
```

Non-standard string literals get the raw, un-interpolated text, which is why regex literals don't need double-escaping. Flags after the closing quote (`r"x"i`) arrive as a second argument.

Generated functions

`@generated` functions run at **compile time**, after types are known but before code is generated, and return an *expression* to be compiled for that specific type signature. Inside, arguments are their **types**, not values.

```
@generated function unrolled_sum(t::NTuple{N, T}) where {N, T}
    ex = :(t[1])
    for i in 2:N
        ex = :($ex + t[$i])
    end
    return ex
end

unrolled_sum((1, 2, 3))    # compiles to literally t[1] + t[2] + t[3]
```

Uses: unrolling loops whose length is in the type (tuples, StaticArrays), generating specialised code from type parameters, avoiding runtime introspection.

Constraints, which are strict:

- You see types only, never values.
- The body must be pure — no side effects, no I/O, no observing global state that might change.
- You cannot call functions defined later, or newly defined methods (no "world age" surprises allowed).
- Debugging is unpleasant.

Almost always, a plain function plus `Val`, or a `@nexprs`-style macro from `Base.Cartesian`, does the job with less pain. Reach for `@generated` last.

Reading and rewriting expressions

Manual walking:

```
function replace_symbol(ex, from, to)
    ex isa Symbol && return ex === from ? to : ex
    ex isa Expr || return ex
    Expr(ex.head, map(a -> replace_symbol(a, from, to), ex.args)...)
end

replace_symbol(:(a + b*a), :a, :z)     # :(z + b*z)
```

`MacroTools.jl` makes this far less tedious and is what most packages use:

```
using MacroTools

@capture(ex, f_(args__))                # destructure by pattern; binds f and args
@capture(ex, function fname_(params__); body__; end)

postwalk(x -> x isa Symbol && x === :a ? :z : x, ex)
prewalk(...)
striplines(ex)                          # drop LineNumberNodes for readable dumps
rmlines(ex)
@q quote ... end                        # quote without LineNumberNodes
```

_ suffixes in `@capture` patterns are the binders; __ slurps a sequence. It's the pattern matching you'd otherwise write by hand.

Reflection at runtime

```
methods(f); which(f, Tuple{Int}); @which f(1)
fieldnames(T); fieldtypes(T); nfields(x)
getfield(x, :a); setfield!(x, :a, 1)
getproperty(x, :a)                      # what x.a lowers to; overloadable
propertynames(x)
isdefined(Main, :x); @isdefined x
names(SomeModule; all = true)
functionloc(f, Tuple{Int})
Base.return_types(f, (Int,))
```

Overloading property access is a legitimate, non-macro form of metaprogramming:

```
struct Lazy
    d::Dict{Symbol,Any}
end
Base.getproperty(l::Lazy, s::Symbol) = getfield(l, :d)[s]
Base.propertynames(l::Lazy) = keys(getfield(l, :d))
```

Note `getfield(l, :d)` inside — using `l.d` would recurse infinitely.

When not to use macros

The honest guidance, given how tempting this all is:

- If a function will do, use a function. Macros don't compose, can't be passed around, can't be `mapped`, and confuse tooling.
- Macros are justified when you need the *unevaluated syntax*: to stringify it (`@show`, `@assert`), to delay or repeat evaluation (`@time`, `@threads`), to rewrite it (`@.`, `@views`), or to generate definitions (`@kwdef`, `@enum`).
- Metaprogramming at load time (a `for` loop with `@eval`) is much safer than a macro and covers most "I have twenty near-identical methods" cases.
- Every macro you write is a small DSL your future self has to remember. The ecosystem's macro-heavy packages are macro-heavy because they're implementing genuinely new syntax (JuMP's constraint algebra, Turing's model notation), not because macros were convenient.

Debugging checklist

```
@macroexpand @mymacro x           # what did it produce?
MacroTools.striplines(@macroexpand @mymacro x)    # readable version
dump(:(the thing I'm matching))   # what does the AST actually look like?
Meta.show_sexpr(ex)               # lisp-style view, sometimes clearer than dump
```

Symptoms and causes:

Symptom	Likely cause
`UndefVarError` for a caller's variable	missing `esc`
Caller's variable mysteriously overwritten	escaped a temporary you introduced
Argument evaluated twice	interpolated `$(esc(ex))` in two places; bind to a gensym
Works at top level, fails in a function	used `eval`, which is module-scoped
"no method matching" on a Symbol	forgot `QuoteNode` where a symbol had to appear as a value