Julia's macros, expressions, etc. for and by the confused

Gray Calhoun

October 22nd, 2014

What are we going to cover?

- How does Julia implement functions and object methods?
- How does Julia represent its own source code?
- What are these “macros” we’ve heard so much about?
    - What do they let us do?
    - How do we write them?
- How do Julia’s macros and functions compare to R’s functions? (time permitting)
- Useful other resources:
    - Julia documentation on Functions, Types, Methods, and Metaprogramming (links are for v0.3)
    - Steven Johnson’s Euro SciPy 2014 presentations (several IJulia notebooks)
    - I wrote a blog post while preparing this talk: http://clhn.org/1019Oym
- re: copying these slides: (since this is going online):

Quick review: what is a function (in Julia)?

- Abstractly, maps a collection of variables to value
  - The collection of variables is a tuple
- Functions are objects (just like in R)
  - You can pass functions as arguments to other functions
- Functions can be named or anonymous
- Other features:
  - Variable numbers arguments
  - Optional arguments with default values
  - Keyword arguments
  - Return multiple values as a tuple

Examples of basic function syntax

```
function f(arg1, arg2)
    if isa(arg1, Int64)
        return arg1
    elseif isa(arg2, Function)
        return arg2(arg1)
    end
    arg1, arg2 # <- implicitly returned
end

f(17., x -> x^2)
#      ========
#> 289.0

g(arg1, arg2, arg3...) = arg1(arg2, arg3)

a, b  = g(17, 329, 932) do x, y # <- anon. fn as
            x, y, map(z -> x*z, y) # first arg.
        end[1:2] # <- discard third value returned
#> (17,(329,932))
```

Rewriting the previous examples with multiple dispatch

```julia
# We would never want to write a function like
# this in Julia:
function f(arg1, arg2)
    if isa(arg1, Int64)
        return arg1
    elseif isa(arg2, Function)
        return arg2(arg1)
    end
    arg1, arg2 # <- implicitly returned
end

# A better way is to define methods
h(x::Int64, y) = x
h(x, g::Function) = g(x) # <- get a warning
h(x, y) = x, y # Fallback definition (the default)

# Should define this as well (first)
h(x::Int64, g::Function) = x
```

Quick discussion of Julia's object system

- Object system is similar to R's
  - Methods "belong" to the generic function, not to the object
  - Discoverability becomes much easier than in (say) Python
  - Naming becomes more important
- *Just in Time* compilation:
  - The first time we call h (or f, for that matter) Julia compiles a version of that function for the datatypes used in the call
  - Future calls with the same datatypes use the precompiled version (which is fast)
- Lots of performance implications
  - Global scope is slow
  - Explicitly defining types *sometimes* matters, but not usually
    - Matters when defining objects, for example
    - Matters for keyword arguments
    - Doesn't matter for normal function arguments
  - "Type-stability" is important
  - More at http://julia.readthedocs.org/en/latest/manual/performance-tips
  - Also see Leah Hanson's TypeCheck.jl package

Expressions and symbols are objects

```
# The ‘Symbol’ object type is used for variable
# names. Symbols start with a ‘:’

:height
#> :height

typeof(:height)
#> Symbol

1 + :height
#> ERROR: ‘+‘ has no method matching +(::Int64, ::Symbol

# Code is represented as an ‘Expr’ object.
:(height + 23)
#> :(height + 23)

Expr(:call, :+, :height, 23)
#> :(height + 23)
```

Evaluating Julia expressions through “eval”

```
# The code is run by evaluating it. This can
# be done manually through ‘eval’

eval(:(height + 23))
#> ERROR: height not defined
eval(:(height = 70))
height
#> 70
eval(:(height + 23))
#> 93

# ‘eval’ always executes in the global (module)
# namespace, not the local namespace. AVOID eval!

f(height) = eval(:(height + 23))
f(12)
#> 93
```

Basic expression facts and syntax

```
# Expressions have two (main) fields
# - head: the ‘type’ of the expression
# - args: the terms that make up the expression

a = :(x < $height < y)
#> :(x < 70 < y)
a.head
#> :comparison
a.args
#> 5-element Array{Any,1}:
#>   :x
#>   :<
#> 70
#>   :<
#>   :y
a.args[3] = :(2 * $height)
a
#> :(x < 2 * 70 < y)
```

Macros are used to programatically manipulate syntax

- It's useful to be able to manipulate Julia expressions and run the new expressions
- It's even more useful if we don't have to do this by hand
- "Macros" are the way to do this in Julia
- A macro is superficially like a function **except**
  - Macros do not evaluate their arguments when they dispatched, they treat their arguments as if they were quoted expressions
  - Macros return expressions: those expressions are then evaluated in the environment that called the macro
  - If a macro is used inside a function, it is executed when the function is defined, before the function is compiled or run
- Since expressions are objects in Julia, our macros are programmed in Julia
  - Similar to Lisp
  - Not at all how C or C++ do macros
  - You can do this with functions in R, which makes programming… interesting

My favorite macro example, slide 1

```
using Devectorize
x = rand(100); y = randn(100)

#macro name
#-----
@devec r = exp(abs(x - y))
#      --------------------
#      Single expression passed to @devec

# @devec does the following:
#   1. writes an expression that has Julia code
#      to define ‘r’
#   2. writes a loop
#      - that iterates down ‘x’ and ‘y’
#      - has ‘r[i] = exp(abs(x[i] - y[i]))’ as
#        its body
# After @devec returns, Julia runs the new expression
```

My favorite macro example, slide 2

```
macroexpand(:(@devec r = exp(abs(x - y))))
# returns (with some editing)
quote
    _siz_16093 = Devectorize.ewise_shape(size(x),size(y)
    if _siz_16093 == () # <- ‘uniqified’ var. names
        _tmp_16092 = exp(abs(x - y))
    else
        _siz_16093 = Devectorize.ewise_shape(size(x),siz
        _ty_16094 = Devectorize.result_type(TFun{:exp}()
        _tmp_16092 = Array(_ty_16094,_siz_16093)
        _len_16095 = length(_tmp_16092)
        for _i_16096 = 1:_len_16095
            _tmp_16092[_i_16096] = # <- assignment
               exp(abs(Devectorize.get_value(x,_i_16096)
                       - Devectorize.get_value(y,_i_1609
        end
    end
    r = _tmp_16092
end
```

General uses of macros

1. Performance
    - @devec, @parallel, @inbounds, @simd, probably more
2. Syntactical “sugar”
3. Extending the language & syntax
    - Keyword arguments were originally introduced as macros in a separate package
    - Docstrings are being added to Base, started as (still is) the Docile.jl package
    - Haskell/Scala style Pattern matching (Match.jl, PatternMatch.jl)
    - Tail-Call Optimization (http://blog.zachallaun.com/post/jumping-julia)
    - Lots of other examples I’m unaware of (probably)
4. Implementing Domain-Specific Languages
    - Distinction between previous bullet not always clear
    - Regular Expressions
    - Regression formulas and DataFrame manipulation (DataFrames.jl, GLM.jl, DataFramesMeta.jl)
    - Optimization (JuMP.jl)
    - etc.

Let's write a (nontrivial) macro!

```
# Dynamic models are annoying to work with!
#
# Say we have an ARMA model

y[t+1] = a0 + a[1]*y[t] + a[2]*y[t-1] + e[t+1] + b*e[t]
e[t+1] ~ Normal(0, v)

# ‘Vectorizing’ this is unpleasant
# Need to be careful about endpoints for loops

# Wouldn’t this be nice?
@loop_ts 500 y[1:2] = (0,0) begin
  y[t+1] = a0 + a[1]*y[t] + a[2]*y[t-1] + e[t+1]+b*e[t]
  e[t] = Normal(0, v)
end
# Let a macro figure out the endpoints, etc.
```

We should start with a baby macro

```
# Let’s leave ‘self initialization,’ ‘robustness,’
# etc. as an optional homework exercise
y = zeros(500)
e = randn(500)

# Start with an example of the syntax we’d like:
@loop_ts y[t+1] = 0.8y[t] + 0.02y[t-2] + e[t+1]

# And the code we want it to generate:
for _t in 3:(length(y) - 1)
    y[_t+1] = 0.8y[_t] + 0.02y[_t-2] + e[_t+1]
end
# Our macro needs to:
# 1. determine which symbols are the vectors
# 2. extract the smallest and largest allowable index
# Other tasks (i.e. the loop body) are easy
```

We should start with a baby macro

```
# Let’s leave ‘self initialization,’ ‘robustness,’
# etc. as an optional homework exercise
y = zeros(500)
e = randn(500)

# Start with an example of the syntax we’d like:
@loop_ts y[t+1] = 0.8y[t] + 0.02y[t-2] + e[t+1]

# And the code we want it to generate:
for _t in 3:(length(y) - 1)
    y[_t+1] = 0.8y[_t] + 0.02y[_t-2] + e[_t+1]
end
# Our macro needs to:
# 1. determine which symbols are the vectors
# 2. extract the smallest and largest allowable index
# Other tasks (i.e. the loop body) are easy
```

Let’s do it live! Juno/LightTable works pretty well

Source code for the baby macro (in case 'live' is a flop) (slide 1)

```
macro loop_ts(ex)
    l, r = ex.args
    idx =
        if isexpr(l.args[2], :call)
            filter(x -> isa(x, Symbol),
                   l.args[2].args[2:end])[1]
        elseif isa(l.args[2], Symbol)
            l.args[2]
        end
    offsets = extrema(vcat(get_offsets(l),
                           get_offsets(r)))
    loopindex = :($(1 - offsets[1]):(length($(l.args[1]))
                                     - $(offsets[2])))
    quote
        for $idx in $loopindex
            $ex
        end
    end
end
```

Source code for the baby macro (in case 'live' is a flop) (slide 2)

```
function get_offsets(ex_::Expr)

    isexpr(ex_,:call) &&
        return [[get_offsets(a)
                 for a in ex_.args[2:end]]...]

    isexpr(ex_,:ref) &&
        return get_offset_from_ref(ex_.args[2])

    warning("Not expecting to be here")
    return Int64[]
end

get_offsets(x) = Int64[]
```

Source code for the baby macro (in case ‘live’ is a flop) (slide 3)

```
get_offset_from_ref(s::Symbol) = 0
get_offset_from_ref(x::Number) = x

function get_offset_from_ref(ex_::Expr)

  if isexpr(ex_,:call)
      ex_.args[1] == :+ &&
          return sum([get_offset_from_ref(a)
                      for a in ex_.args[2:end]])

      ex_.args[1] == :- &&
          return (get_offset_from_ref(ex_.args[2])
                  - sum([get_offset_from_ref(a)
                         for a in ex_.args[3:end]]))
  end
  warning("Didn’t expect to get here")
  return(0)
end
```

Syntax for defining and calling macros

```
# we define a macro like this:
macro mymacro(e1, e2, e3)
  # syntax-y stuff here
end

# mymacro can be called in two ways:
@mymacro(expr_1, expr_2, expr_3) # <- no space before
# or                                    ‘(’ !!!
@mymacro expr_1 expr_2 expr_3

# We can also define a macro
macro m_str(p) # <- p is now going to be a string
  # syntax-y stuff here
end

# m_str gets called as
m"RU 1337 H4X0RZ!?!" # <- clearly a DSL
```

Additional points to consider

- You can override gensym by using the ‘esc’ function.
- This lets you define new variables inside the macro that you can refer to after the macro ends

John Myles White suggested contemplating these questions:

1. When is a function or macro evaluated?
    1.1. Bonus: when are the inputs to a function or macro evaluated?
2. What are the types of its inputs?
3. What are the types of its outputs?

Just to confuse you more:

- Functions can take quoted expressions as arguments!
- But those arguments generally **should not be evaluated**
    - ‘eval’ doesn’t let you access local scope
    - It’s very slow

Comparison with R

- R's functions are probably **more** flexible than Julia's methods + macros system
    - *Environments* are first-class variables
    - 'eval' in R can take place in local scope, or in arbitrary other scopes
    - Argument names can be captured by functions by quoting inside the function
- It's not clear that we lose any expressive power in Julia
- R's flexibility may be useful when doing interactive data analysis
    - R's metaprogramming does seem underused
    - Definitely could be an avenue for introducing bugs when you build complicated software that relies on it
    - Makes it difficult for the interpreter/compiler to write efficient code (at least, with existing tech)
    - e.g.:

```r
# What happens here?
x <- 3; y <- 3

f(x); f(y); f(z <- 3); f(3)
```

Last slide, I'll fake wisdom and drop some knowledge

- Incremental development
  - Of individual macros
  - Of collections of macros
- Embedded scientific DSL have huge potential, should be very exciting
  - In econ, "Dynare" has been massively popular, transformed macroeconomics
    - It's written in **Matlab**
  - The S statistical formula notation is popular
  - ggplot is essentially a DSL for statistical graphics
- Balance, incremental growth
- Expect lots of changes in Julia over 0.4 and 0.5 release
  - Macro hygiene
  - Potentially moving Expr to an abstract type
- Documentation and tutorials are pretty barren
  - Look at individual packages that rely on macros
  - Look at the **first release** of those packages
  - Lisp books: Paul Graham's *On Lisp* and Doug Hoyte's *Let over Lambda*