Chapter 17: Metaprogramming, Big Picture

Tony ElHabr

R4DS Reading Group

What is metaprogramming?

What is metaprogramming?

What is metaprogramming?

What is metaprogramming?

What is metaprogramming?

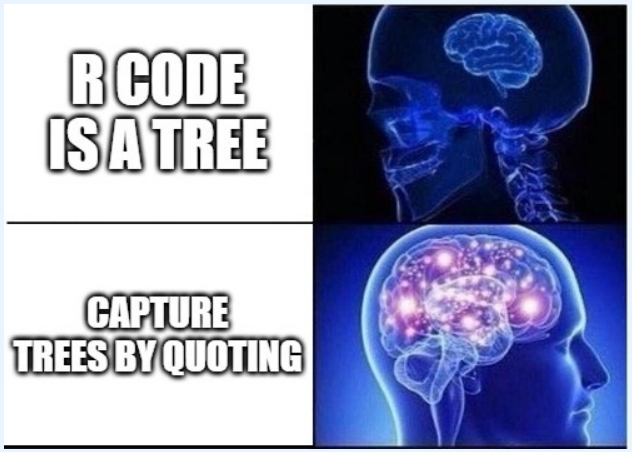
Writing programs (or code) that can manipulate other programs (or code).

Does metaprogamming == Non Standard Evaluation (NSE)?

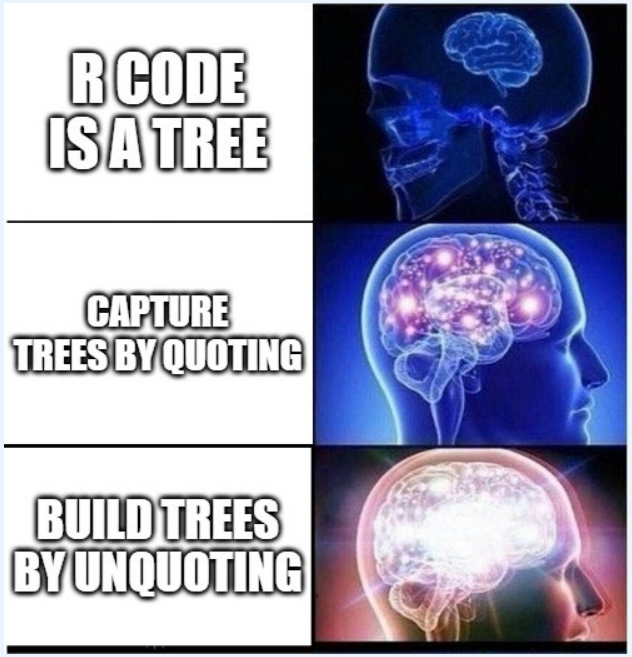
> [Metaprogramming] is the idea that code is data that can be inspected and modified programmatically... Closely related to metaprogramming is non-standard evaluation, NSE for short. This term, which is commonly used to describe the behaviour of R functions, is problematic.. NSE is actually a property of the argument (or arguments) of a function, so talking about NSE functions is a little sloppy.

Big Ideas

Big Ideas



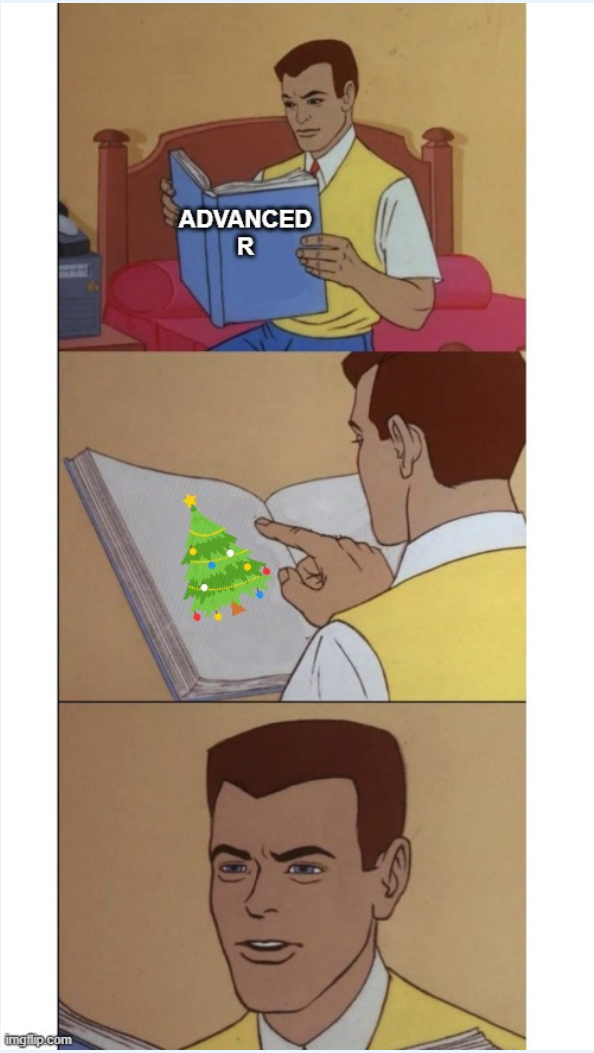
Big Ideas


R CODE
IS A TREE
CAPTURE
TREES BY QUOTING
BUILD TREES
BY UNQUOTING

Big Ideas

References: (1) https://rstudio.com/resources/rstudioconf-2018/tidy-eval-programming-with-dplyr-tidyr-and-ggplot2/, (2) https://www.youtube.com/watch?v=nERXS3ssntw, (3) https://www.youtube.com/watch?v=g1h-YDWVRLc

1. R code is a tree



1. R code is a tree

But what does that mean???

"R code can be described as expressions, which can be drawn as trees."

> An expression is anything that has a value. The simplest expressions are literal values like the number 1, the string "stuff", and the Boolean TRUE. A variable like least is also an expression: its value is whatever the variable currently refers to. Complex expressions are built out of simpler expressions: `1 + 2` is an expression that uses `+` to combine 1 and 2, while the expression `c(10, 20, 30)` uses the function `c` to create a vector out of the values 10, 20, 30. Expressions are often drawn as trees.

```
lobstr::ast(f(x, 'y', 1))
```

```
## o-f
## +-x
## +-"y"
## \-1
```

> Colours will be shown when you call `ast()`, but do not appear in the book for complicated technical reasons.

https://third-bit.com/2018/11/16/non-standard-evaluation.html

1. R code is a tree

But what does that mean???

"R code can be described as expressions, which can be drawn as trees."

> An expression is anything that has a value. The simplest expressions are literal values like the number 1, the string "stuff", and the Boolean TRUE. A variable like least is also an expression: its value is whatever the variable currently refers to. Complex expressions are built out of simpler expressions: `1 + 2` is an expression that uses `+` to combine 1 and 2, while the expression `c(10, 20, 30)` uses the function `c` to create a vector out of the values 10, 20, 30. Expressions are often drawn as trees.

```
lobstr::ast(f(x, 'y', 1))
```

```
## o-f
## +-x
## +-"y"
## \-1
```

> Colours will be shown when you call `ast()`, but do not appear in the book for complicated technical reasons.

https://third-bit.com/2018/11/16/non-standard-evaluation.html

1. R code is a tree

Everything is a tree!

Assignment and infix operator (*)

```
x <- 1
lobstr::ast(y <- 2 * x)
```

```
## o-`<-`
## +-y
## \-o-`*`
##   +-2
##   \-x
```

Control flow statements

```
lobstr::ast(if(x > 1) y else x)
```

```
## o-`if`
## +-o-`>`
## | +-x
## | \-1
## +-y
## \-x
```

Functions

```
lobstr::ast(function(x, y) x + y)
```

```
## o-`function`
## +-o-x = ``
## | \-y = ``
## +-o-`+`
## | +-x
## | \-y
## \-<inline srcref>
```

ASTs

```
lobstr::ast(lobstr::ast(x + y))
```

```
## o-o-`::`
## | +-lobstr
## | \-ast
## \-o-`+`
##   +-x
##   \-y
```

2. Capture trees by quoting

```
ex1 <- rlang::expr(x + y)
ex1
```

```
## x + y
```

```
x <- 1
y <- 2
eval(ex1)
```

```
## [1] 3
```

```
lobstr::ast(1 + 2)
```

```
## o-`+`
## +-1
## \-2
```

2. Capture trees by quoting

`rlang::expr` vs. `rlang::enexpr`

`rlang::expr` quotes your expression

```
f1 <- function(z) expr(z)
f1(a + b)
```

```
## z
```

`enexpr` quotes user's expression

```
f2 <- function(z) enexpr(z)
f2(a + b)
```

```
## a + b
```

`en` = "enriched"

2. Capture trees by quoting

2. Capture trees by quoting


```
mean(x + y)

library(ggplot2)

# ggplot(mtcars, aes(disp, mpg)) + geom_point()

mtcars$disp

z <- x + 1

data.frame(z = 3)
```

Blue: Evaluated using usual R rules

Red: Quoted and evaluated with special rules

3. Build trees by unquoting


```
lobstr::ast(eval(!!ex1))
```

```
## o-eval
## \-o-`+`
##   +-x
##   \-y
```

```
ex2 <- rlang::expr(x / !!ex1)
ex2
```

```
## x/(x + y)
```

```
eval(ex2)
```

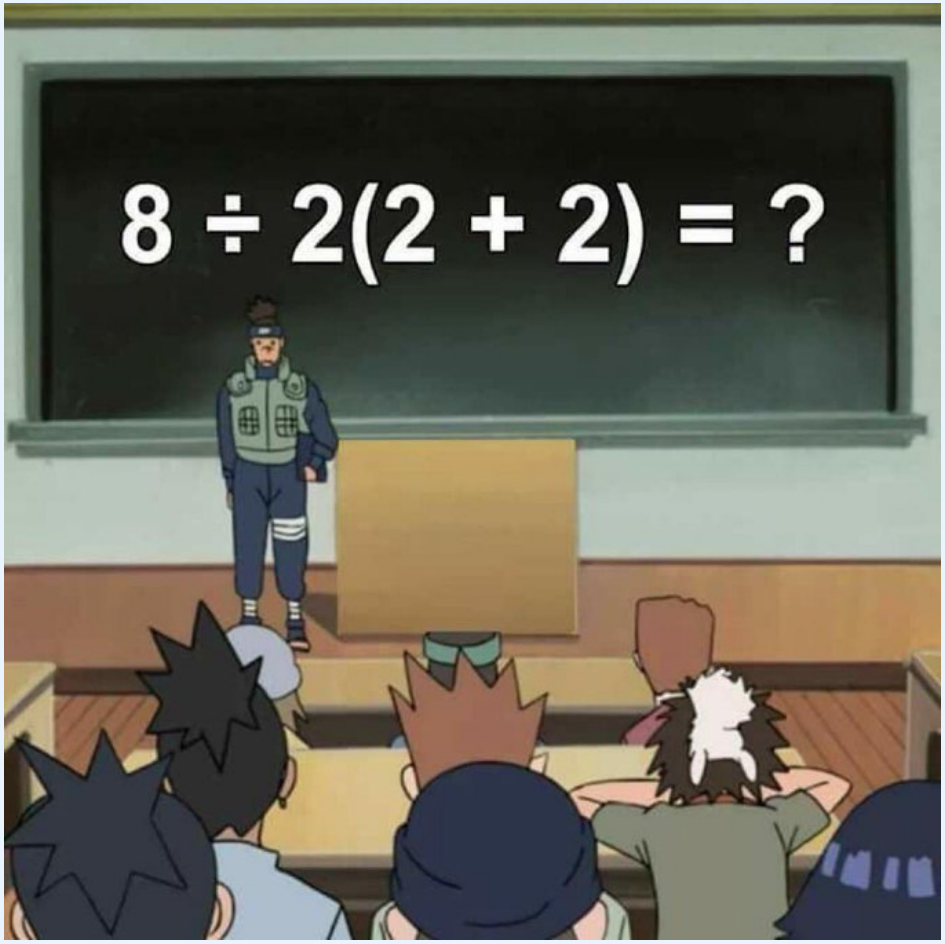
```
## [1] 0.3333333
```

```
lobstr::ast(eval(!!ex2))
```

```
## o-eval
## \-o-`/`
##   +-x
##   \-o-`+`
##     +-x
##     \-y
```

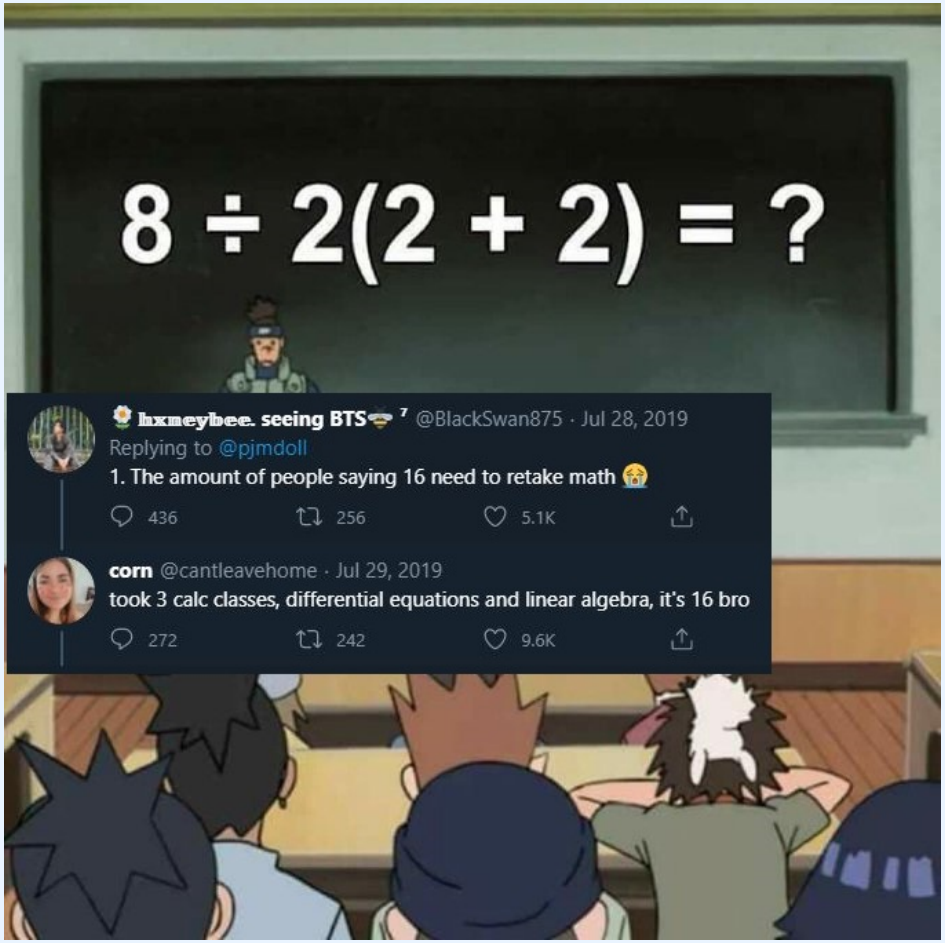
3. Build trees by unquoting

Understanding how to build code trees == success in online arguments.



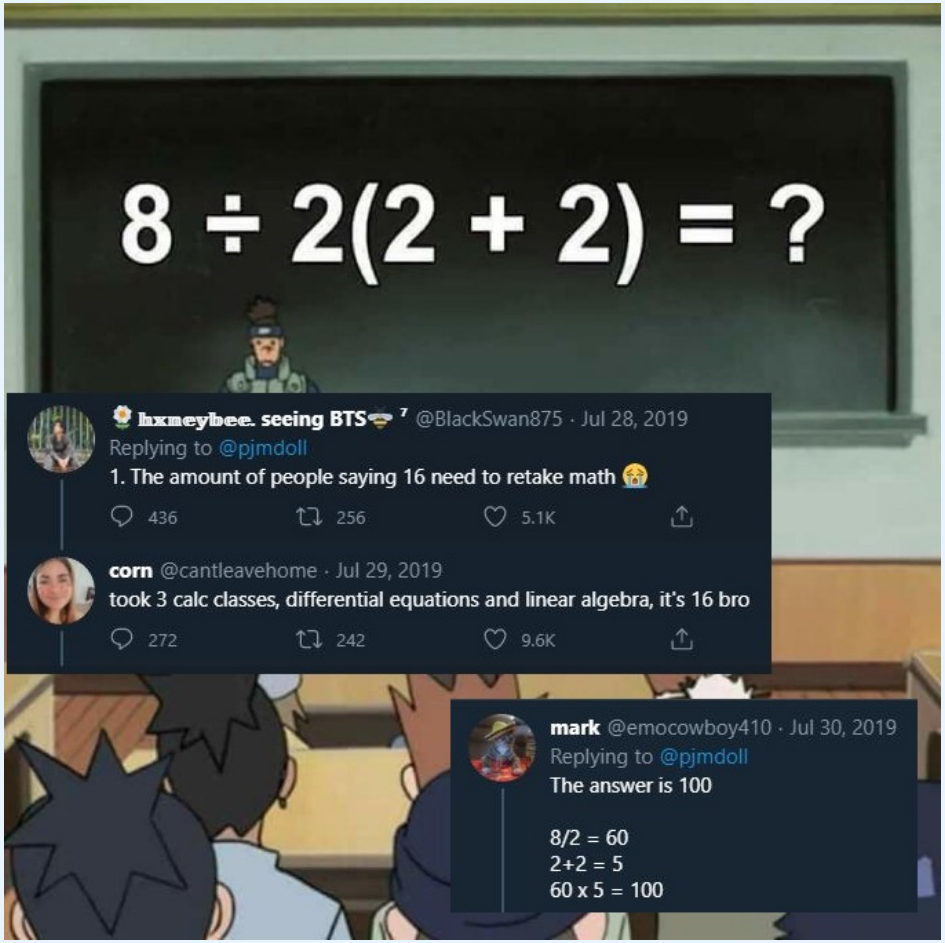
3. Build trees by unquoting

Understanding how to build code trees == success in online arguments.



3. Build trees by unquoting

Understanding how to build code trees == success in online arguments.



3. Build trees by unquoting

Understanding how to build code trees == success in online arguments.

```
x <- 8 / 2 * (2 + 2)

lobstr::ast(x <- 8 / 2 * (2 + 2))
```

```
## o-`<-`
## +-x
## \-o-`*`
##   +-o-`/`
##   | +-8
##   | \-2
##   \-o-`(`
##     \-o-`+`
##       +-2
##       \-2
```

```
x
```

```
## [1] 16
```

4. Capture trees AND environments

quosure == closure + quote

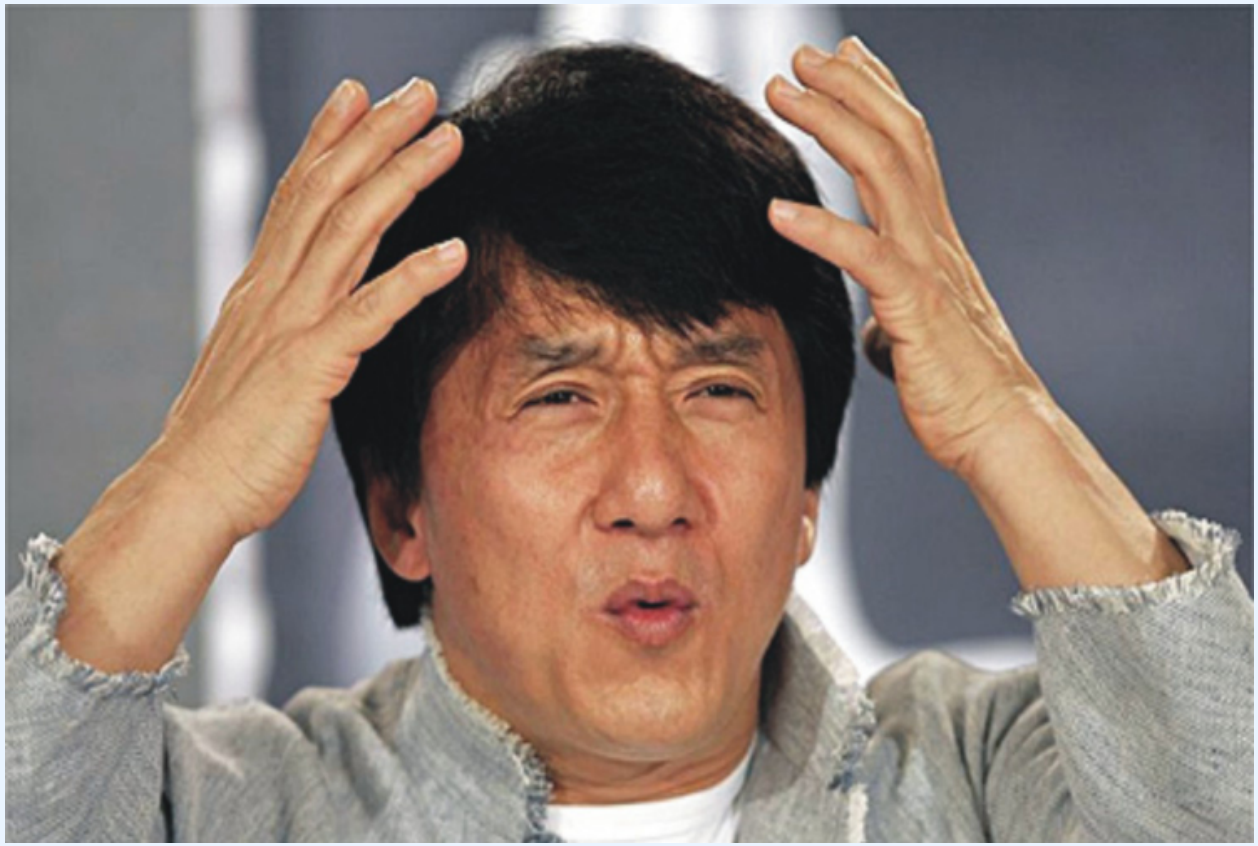
Quosures aremportant for disambiguating the context in which expressions are evaluated (e.g. a column in a data frame or a variable in the parent environment).

4. Capture trees AND environments

Result is `tibble(x = 0, y = 1)`.

```
update <- function(df, col) {
  n <- 1
  col <- rlang::enexpr(col)
  res <- dplyr::mutate(df, y = !!col)
  res
}

df <- tibble::tibble(x = 0)
n <- 2
update(df, x + n)
```

4. Capture trees AND environments

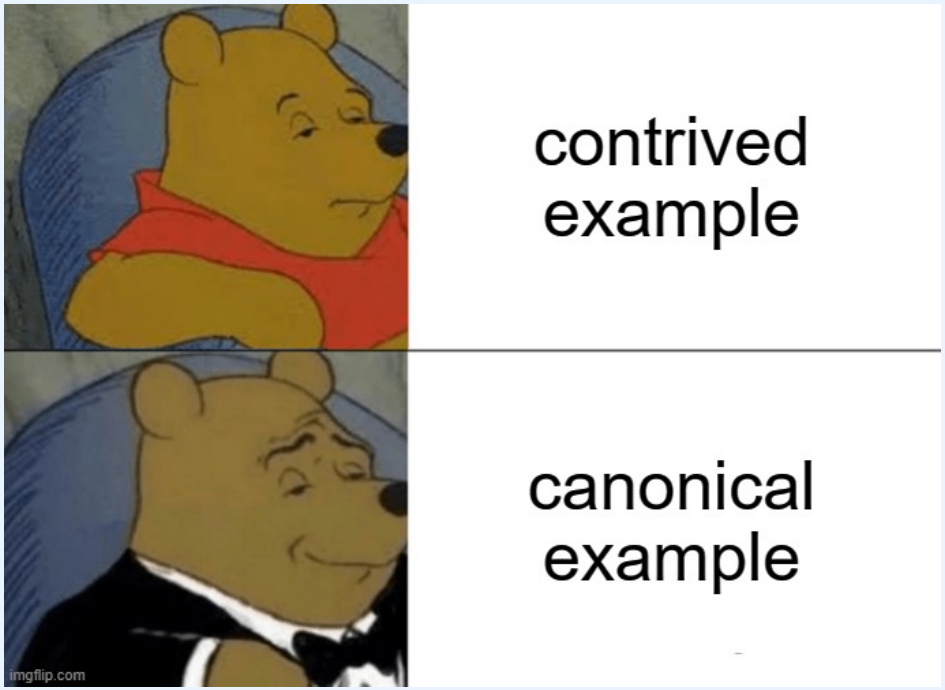


4. Capture trees AND environments

Result is `tibble(x = 0, y = 2)`.

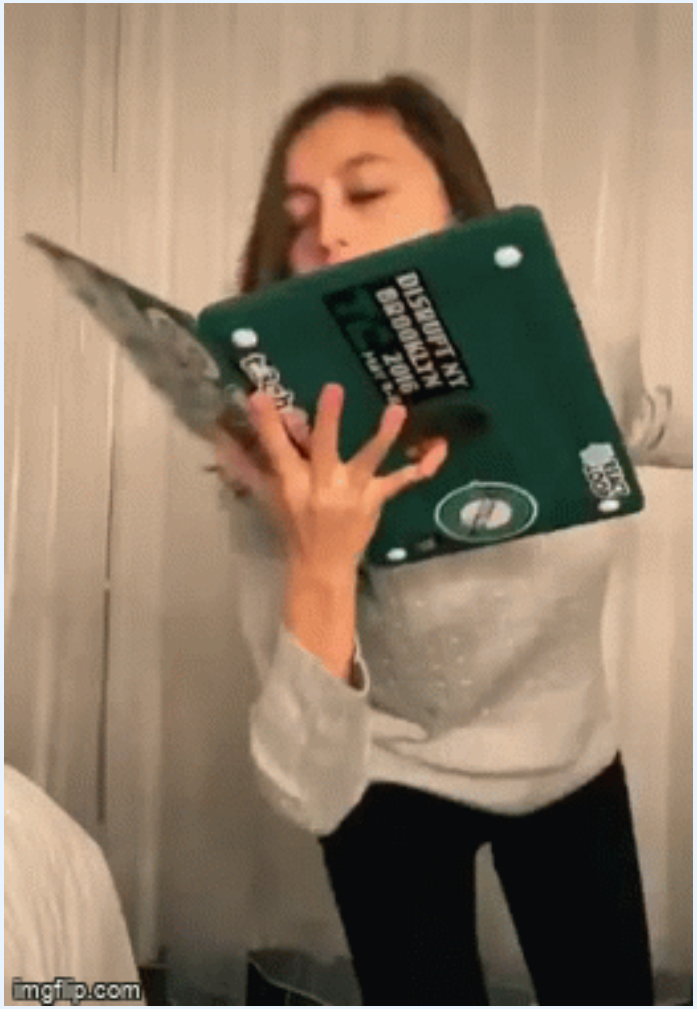
```
update <- function(df, col) {
  n <- 1
  col <- rlang::enquo(col)
  res <- dplyr::mutate(df, y = !!col)
  res
}

df <- tibble::tibble(x = 0)
n <- 2
update(df, x + n)
```

4. Capture trees AND environments



The Power of Metaprogramming

Metaprogramming is awesome. R is great because of how much control it allows user to have.

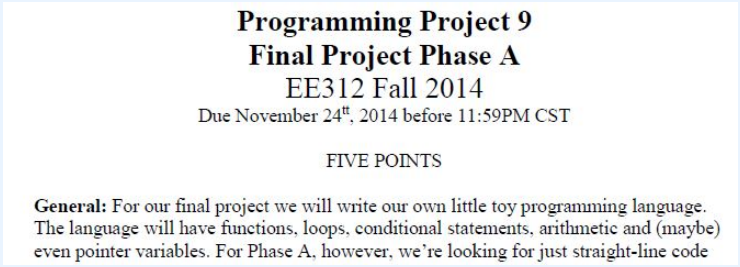


What's next

- Chapter 18: More about R code as a tree
- Chapter 19: More about evaluating (quoting) unevaluated code
- Chapter 20: More about evaluating (unquoting) captured code

Aside about trees

Understanding how "code is a tree" would have helped me with my final project in my Intro to Programming class.



Programming Project 9
Final Project Phase A
EE312 Fall 2014
Due November 24th, 2014 before 11:59PM CST

FIVE POINTS

General: For our final project we will write our own little toy programming language. The language will have functions, loops, conditional statements, arithmetic and (maybe) even pointer variables. For Phase A, however, we're looking for just straight-line code

> Please note that whether you choose to explicitly build a parse tree or not, you will almost certainly have to write your parsing and executing function(s) using recursion. Fortunately, the recursion required to do this is super easy (whether you build a parse tree or not, the recursion is very natural).

> Implementing a Parse Tree is optional for Phase A. In Phase B, we will almost certainly mandate that you have a Parse Tree, and for Phase B, the parse trees are decidedly more complicated, since you will have both Expressions to represent and Statements to represent.

Aside about trees

```
**********************************************************************
TEST:         Test07
DESCRIPTION:  Light functions - no parameters
WEIGHT:       0.375
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test08
DESCRIPTION:  Functions with one parameter
WEIGHT:       0.375
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test09
DESCRIPTION:  Functions with no return statement
WEIGHT:       0.340
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test10
DESCRIPTION:  Functions with local vars - testing scoping
WEIGHT:       0.330
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test11
DESCRIPTION:  Other local scoping tests
WEIGHT:       0.330
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test12
DESCRIPTION:  More interesting functions - recursion
WEIGHT:       0.340
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test13
DESCRIPTION:  Additional functions test
WEIGHT:       0.330
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
TEST:         Test14
DESCRIPTION:  Nested function calls
WEIGHT:       0.330
RESULT:       FAILED
REASON:       TEST CRASHED
**********************************************************************
```