Chapter 17-20: Metaprogramming

Ryan Heslin

May 30, 2022

18. Expressions

Abstract Syntax Trees

1.

```
f(g(h))
1 + 2 + 3
(x + y) *z
```

2.

```
library(lobstr)

ast(f(g(h(i(1, 2, 3)))))
ast(f(1, g(2, h(3, i()))))
ast(f(g(1, 2), h(3, i(4, 5))))
```

3.

The first just uses backticks, which aren't quoted. In the second, `**` is an alias for `^`. In the third, right assign is an alias for left assign with the argument order swapped.

4.

`srcref` indicates an inlined expression within another, here the empty body of the created function. We also see how the formals and body components are separated by the `function` function.

5.

The subsequent `else`s are nested within the previous `if`, since `if...else` is syntactic sugar for repeated calls of the `if` function.

```
ast(
  if (x < 4) {
    y
  } else if (x < -2) {
    z
  } else if (q) {
    d
  } else {
    v
  }
)
```

Parsing and Grammar

1.

It can contain both constants and calls. At the top level, it prints output explicitly.

```
ast(f((1)))
ast(`(`(1 + 1))
```

2.

It should not be used for assignment.

```
mean(x = 1:10)
```

```
[1] 5.5
```

```r
x <- 1:10
```

3.

-4, because ^ has precedence over -.

```r
-2^2
```

```
[1] -4
```

4.

The outer `!` modifies the whole expression. The inner expression resolves to 1 because `FALSE` coerces to a numeric 0, and coercing 1 to logical and negating gives `FALSE`. So the result is `FALSE`.

```r
ast(!1 + !1)
```

5.

The chaining works because `<-` returns its right-hand side, and lazy evaluation ensures no name is evaluated before it is bound.

```r
x1 <- x2 <- x3 <- 0
```

6.

They beat unary but not binary operators.

```r
ast(x + y %+% z)
ast(x^y %+% z)
```

7.

Sensibly, it throws an error.

```r
try(parse_expr("x + 1; y + 1"))
```

```
Error in parse_expr("x + 1; y + 1") :
  `x` must contain exactly 1 expression, not 2.
```

8.

"Unexpected end of input."

```r
try(parse_expr("a +"))
```

```
Error in parse(text = elt) : <text>:2:0: unexpected end of input
1: a +
   ^
```

9.

It returns parsed lines of a given width. But the lifecycle is questioning!

```r
expr_text(lm(data = mtcars, formula = cyl ~ .))
```

```
[1] "structure(list(coefficients = c(`(Intercept)` = 12.1071985479796, \nmpg = -0.0048566471993194, disp
```

10.

To my surprise, it doesn't break.

```r
pairwise.t.test(Lorem <- ipsum <- dolor <- sit <- ametconsetetur <- sadipscing <- elitrsed <- diam <- no
```

```
    Pairwise comparisons using t tests with pooled SD

data:  Lorem <- ipsum <- dolor <- sit <- ametconsetetur <- sadipscing <- elitrsed <- diam <- nonumy <- e

   0 1 2 3 4 5 6 7 8 9
1  - - - - - - - - - - -
2  - - - - - - - - - - -
```

```
3  - - - - - - - - - - -
4  - - - - - - - - - - -
5  - - - - - - - - - - -
6  - - - - - - - - - - -
7  - - - - - - - - - - -
8  - - - - - - - - - - -
9  - - - - - - - - - - -
10 - - - - - - - - - - -

P value adjustment method: holm
```

Expressions

1.

Raw and complex, which have constructor functions and so can't be entered directly as constants.

2.

It turns into a call with the new first element as the outermost function, which often means the result is nonsensical.

```
library(rlang)

x <- expr(read.csv("foo.csv", header = TRUE))
x[-1]
```

```
"foo.csv"(header = TRUE)
```

```
(x[-1])[-1]
```

```
TRUE()
```

3.

The first unquotes all call components. The second quotes `median`, deferring method dispatch. The third quotes `x`, so the generic delegates to `UseMethod` but `x` isn't substituted. The fourth quotes both elements, leaving the call as entered.

```r
x <- 1:10

call2(median, x, na.rm = TRUE)
```

```
(function (x, na.rm = FALSE, ...)
UseMethod("median"))(1:10, na.rm = TRUE)
```

```r
call2(expr(median), x, na.rm = TRUE)
```

```
median(1:10, na.rm = TRUE)
```

```r
call2(median, expr(x), na.rm = TRUE)
```

```
(function (x, na.rm = FALSE, ...)
UseMethod("median"))(x, na.rm = TRUE)
```

```r
call2(expr(median), expr(x), na.rm = TRUE)
```

```
median(x, na.rm = TRUE)
```

4.

`mean` passes `na.rm` via dots, and it is only matched in `mean.default`, not the `mean` generic, so the function can't standardize the call.

5.

`foo` is a function being called, not a tagged argument, so it needs no name, and indeed the `names` attribute only exists in the call object and will not impact the call when executed

```r
x <- expr(foo(x = 1))
names(x) <- c("x", "y")
x
```

```
foo(y = 1)
```

6.

The arguments to `call2` appear in the order of the code: test, if, else.

```
greater <- call2(quote(`>`), quote(x), 1)
call2(quote(`if`), greater, "a", "b")
```

```
if (x > 1) "a" else "b"
```

AST Walking

I added the case for closures.

```
expr_type <- function(x) {
  if (rlang::is_syntactic_literal(x)) {
    "constant"
  } else if (is.symbol(x)) {
    "symbol"
  } else if (is.call(x)) {
    "call"
  } else if (is.pairlist(x)) {
    "pairlist"
  } else if (is.integer(x) && "srcfile" %in% names(attributes(x))) {
    "closure"
  } else {
    typeof(x)
  }
}

expr_type(expr("a"))
```

```
[1] "constant"
```

```
expr_type(expr(x))
```

```
[1] "symbol"
```

```
expr_type(expr(f(1, 2)))
```

```
[1] "call"
```

```
switch_expr <- function(x, ...) {
  switch(expr_type(x),
    ...,
    stop("Don't know how to handle type ", typeof(x), call. = FALSE)
  )
}

logical_abbr_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T"),

    # Recursive cases
    call = ,
    pairlist = purrr::some(x, logical_abbr_rec)
  )
}

logical_abbr <- function(x) {
  logical_abbr_rec(enexpr(x))
}

flat_map_chr <- function(.x, .f, ...) {
  purrr::flatten_chr(purrr::map(.x, .f, ...))
}
```

Modified for problem 3.

```
assignment_fns <- sapply(paste("package", c("base", "stats", "methods"),
  sep
  = ":"
), ls, pattern = "[a-zA-Z_.]+<-$") |>
  unlist(use.names = FALSE) |>
  gsub(pattern = "<-$", replacement = "")

find_assign <- function(x) unique(find_assign_rec(enexpr(x)))

find_assign_call <- function(x) {
  if (is_call(x, "<-") && is_symbol(x[[2]])) {
    lhs <- as_string(x[[2]])
    children <- as.list(x)[-1]
  } else if (is_call(x, "<-") && is.call(x[[2]]) &&
    is_symbol(x[[c(2, 1)]]) && as_string(x[[c(2, 1)]]) %in% assignment_fns &&
    is_symbol(x[[3]])) {
    lhs <- as_string(x[[3]])
    children <- as.list(x)[-c(1, 2)]
  } else {
    lhs <- character()
    children <- as.list(x)
  }
```

```
  c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = ,
    symbol = character(),

    # Recursive cases
    pairlist = flat_map_chr(x, find_assign_rec),
    call = find_assign_call(x)
  )
}

find_assign(a <- b <- c)
```

```
[1] "a" "b"
```

```
find_assign(names(x) <- "b")
```

```
character(0)
```

```
find_assign(names(x) <- b <- "c")
```

```
[1] "b"
```

1, 2.

Seems to work…

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T"),

    # Recursive cases
    closure = logical_abbr_rec(c(formals(x), body(x))),
    call = ,
    pairlist = purrr::some(if (is.symbol(x[[1]]) &&
      as_string(x[[1]]) %in% c("T", "F")) {
      x[-1]
    } else {
      x
```

```
    }, logical_abbr_rec)
  )
}
logical_abbr(T(1))
```

```
[1] FALSE
```

```
logical_abbr(T(1, 2, T))
```

```
[1] TRUE
```

```
logical_abbr(function(x = TRUE) {
  g(x + T)
})
```

```
[1] TRUE
```

```
logical_abbr(function(x = TRUE) {
  g(x + T(1))
})
```

```
[1] FALSE
```

```
logical_abbr(function(x = TRUE) {
  g(x + T(1) + T)
})
```

```
[1] TRUE
```

3.

See `find_assign` modified above.

4.

```
find_fn <- function(x, target) {
  x <- enexpr(x)
```

```
  target <- deparse(enexpr(target))
  tryCatch(
    {
      match.fun(target)
      find_fn_rec(x, target)
    },
    error = function(e) stop(e)
  )
}

find_fn_rec <- function(x, target) {
  switch_expr(x,
    # Base cases
    constant = ,
    symbol = character(),

    # Recursive cases
    pairlist = flat_map_chr(x, find_fn_rec, target = target),
    call = {
      if (name <- as_string(x[[1]]) == target) {
        deparse(x)
      } else {
        flat_map_chr(x, find_fn_rec, target = target)
      }
    }
  )
}
find_fn(sum(1:10, sum(mtcars)), sum)
```

```
[1] "sum(1:10, sum(mtcars))"
```

```
find_fn(
  {
    sum(1:10)
    x + sum
    sum()
  },
  sum
)
```

```
[1] "sum(1:10)" "sum()"
```

```
find_fn(lm(cyl ~ disp, data = mtcars), mean)
```

```
character(0)
```

19. Quasiquotation

Motivation

1.

Quoted:

- `MASS`
- `cyl == 4`
- `sum(vs)`

Unquoted:

- `mtcars`
- `mtcars2$am`
- `mtcars2` (twice)

2.

Quoted:

- `dplyr`
- `ggplot2`
- `cyl`
- `mean(mpg)`
- `cyl` and `mean` in the second call

Unquoted: * * `mtcars` * `mpg` * `aes(cyl, mean)`

Quoting

1.

It's just an `enexpr` wrapper.

2.

Only `f2` substitutes its arguments into the expression. `f1` just quotes the expression.

3.

The "arg must be a symbol" error.

```
enexpr(x + y)
```

```
Error in `enexpr()`:
! `arg` must be a symbol
```

```
enexpr(missing_arg())
```

```
Error in `enexpr()`:
! `arg` must be a symbol
```

4.

`exprs(a)` is an unnamed list with the symbol `a`, `exprs(a=)` a named list of the missing symbol.

5.

`exprs` offers options to force naming and ignore empty arguments.

6.

```
substitute(x + y)
```

```
foo(y = 1) + y
```

```
substitute(x + y, env = list(x = 4))
```

```
4 + y
```

```
foo <- function(x) {
  substitute(x)
}
foo(7)
```

```
[1] 7
```

Unquoting

1.

I had to look up the correct way to parenthesize an unquoted expression component.

```
library(rlang)
xy <- expr(x + y)
xz <- expr(x + z)
yz <- expr(y + z)
abc <- exprs(a, b, c)

expr(!!xy / !!xz)
```

```
(x + y)/(x + z)
```

```
expr(-(!!xz)^!!yz)
```

```
-(x + z)^(y + z)
```

```
expr(((!!xy)) + !!yz - !!xy)
```

```
(x + y) + (y + z) - (x + y)
```

```
expr(atan2(!!xy, !!yz))
```

```
atan2(x + y, y + z)
```

```
expr(sum(!!xy, !!xy, !!yz))
```

```
sum(x + y, x + y, y + z)
```

```
expr(sum(!!!abc))
```

```
sum(a, b, c)
```

```
expr(mean(c(!!!abc), na.rm = TRUE))
```

```
mean(c(a, b, c), na.rm = TRUE)
```

```
expr(foo(!!!exprs(a = !!xy, b = !!yz)))
```

```
foo(a = x + y, b = y + z)
```

2.

`a`, which doesn't force the `:` call ahead of time, is more natural, since `1:10` is only evaluated after `mean` is called.

```
(a <- expr(mean(1:10)))
```

```
mean(1:10)
```

```
(b <- expr(mean(!!(1:10))))
```

```
mean(1:10)
```

```
identical(a, b)
```

```
[1] FALSE
```

```
expr_print(a)
```

```
mean(1:10)
```

```
expr_print(b)
```

```
mean(<int: 1L, 2L, 3L, 4L, 5L, ...>)
```

...

1.

This implementation takes function arguments via dots and captures them using `list2`, which enables values to be defined with expressions in the caller environment, e.g. `!!key := val`. It allows the user to pass arguments to the function via a mix of standard-evaluated arguments, unquoted expressions, and unquoted lists of expressions, and control the evaluation environment.

2.

`interaction` uses the simple `list(...)` and iterates over the result. `par` does the same, but substitutes the global variables `.Pars` if none are passed. Otherwise, it simplifies args to a character vector if possible, then simplifies if there is only one argument. `expand.grid` captures the dots and returns an empty data frame if none are present. It then processes the args list in sequence, generating the expanded vectors.

3.

`x=10` gets matched to the `x` argument, shoving the vector intended to match `x` into `...`.

```
set_attr <- function(x, ...) {
  attrs <- rlang::list2(...)
  attributes(x) <- attrs
  x
}
tryCatch(set_attr(1:10, x = 10), error = function(e) message(as.character(e)))
```

Case Studies

1.

The first approach splices the two arguments into an `x + y` template, while the second uses `call2` to generate the reductions. I think the first form makes its intent clearer but is less "clean."

```
library(purrr)
linear <- function(var, val, use_call2 = FALSE) {
  var <- ensym(var)
  coef_name <- map(seq_along(val[-1]), ~ expr((!!var)[[!!.x]]))

  summands <- map2(val[-1], coef_name, ~ expr((!!.x * !!.y)))
  summands <- c(val[[1]], summands)
  if (use_call2) {
    reduce(summands, call2, .fn = "+")
  } else {
    reduce(summands, ~ expr(!!.x + !!.y))
  }
}
linear(x, c(10, 5, -4))
```

```
10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

```
linear(x, c(10, 5, -4), use_call2 = TRUE)
```

```
10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

2.

```
bc <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x^lambda - 1) / lambda
  }
}

bc2 <- function(lambda) {
  if (lambda == 0) {
    new_function(exprs(x = ), expr(log(x)))
  } else {
    new_function(exprs(x = ), expr((x^!!exec(`-`, lambda, 1)) / !!lambda))
```

```
  }
}
```

3.

```
compose <- function(f, g) {
  function(...) f(g(...))
}

compose2 <- function(f, g) {
  f <- ensym(f)
  g <- ensym(g)
  new_function(exprs(... = ), expr((!!f)((!!g)(...))))
}
compose2(f, g)
```

```
function (...)
f(g(...))
<environment: 0x559c9e1c92b8>
```

20. Evaluation

1.

`source` defaults to the global environment. `local = TRUE` uses the global environment, and an environment name can also be supplied.

2.

Outer `eval` returns a value, outer `expr` an expression. So 4, 4, and a hideous quoted expression.

3.

```
x <- new.env()
x$y <- 5
```

```r
y <- 7

get2 <- function(name, env) {
  name <- sym(name)
  eval(name, envir = env)
}

assign2 <- function(name, value, env) {
  name <- sym(name)
  exp <- expr(`<-`(!!name, !!value))
  eval(exp, env)
}
get2("y", x)
```

```
[1] 5
```

```r
get2("y", globalenv())
```

```
  [1] 16  1 13 21  6 17  9  7 23 19 22 18 26 16 11
 [16] 30 10  7 28 19 29  2 10  1 11 27 26 15 26 29
 [31] 24 10 16 21 12  7 24 21 28 22  8  1 19 26  3
 [46] 11  1 29 14 22  6 28 20  9  5 29 14 17  3 22
 [61]  3  7 14  3 19 12 17  9 28 18 24 22 17 18 23
 [76] 15 27  7 15 20 25 24 19 21 16 19  4  2 16  1
 [91] 12  7  9 28 26 13 26  2  2 19
```

```r
assign2("z", 7, x)
x$z
```

```
[1] 7
```

```r
source2 <- function(path, env = caller_env()) {
  file <- paste(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)

  res <- lapply(exprs, eval, env)
  invisible(lapply(res, print))
}
source2(here::here("misc/source_file.R"))
```

```
[1]  6 12
[1] 5
[1] 1 7
[1]  6 12

Call:
```

```
lm(formula = mpg ~ disp, data = mtcars)

Coefficients:
(Intercept)         disp
   29.59985     -0.04122
```

```
test <- function() {
  test2 <<- function() x
}
```

5.

`expr` is evaluated in `envir`, by default an empty child of the current, and wrapped in a `substitute` call. This call is then evaluated in the caller environment.

Quosures

1.

1, 11, 111.

```
q1 <- new_quosure(expr(x), env(x = 1))
q1
```

```
<quosure>
expr: ^x
env:  0x559cbd39add0
```

```
#> <quosure>
#> expr: ^x
#> env:  0x7fac62d19130
```

```
q2 <- new_quosure(expr(x + !!q1), env(x = 10))
q2
```

```
<quosure>
expr: ^x + (^x)
env:  0x559cbac77da0
```

```
#> <quosure>
#> expr: ^x + (^x)
```

```
#> env:  0x7fac62e35a98
```

```
q3 <- new_quosure(expr(x + !!q2), env(x = 100))
q3
```

```
<quosure>
expr: ^x + (^x + (^x))
env:  0x559cb91d1a10
```

```
#> <quosure>
#> expr: ^x + (^x + (^x))
#> env:  0x7fac6302feb0
```

```
eval_tidy(q1)
```

```
[1] 1
```

```
eval_tidy(q2)
```

```
[1] 11
```

```
eval_tidy(q3)
```

```
[1] 111
```

2.

```
enenv <- function(arg) {
  arg <- enquo(arg)
  quo_get_env(arg)
}
x <- 3
enenv(x)
```

```
<environment: 0x559c9edd7978>
```

```
with(
  mtcars,
  enenv(cyl)
)
```

```
<environment: 0x559c95766120>
```

Data Masks

1.

As in the example, it's possible for a variable to be defined based on an earlier modification of the same variable passed via `...`. That makes vectorization impossible.

2.

This version inlines `rows` into the call to `[` and then evaluates it in the data mask instead of evaluating `rows` in the data mask first and calling `[` directly.

3.

Unquoting `.na_last` prevents the unlikely case of a variable by that name masking the argument.

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
  # quote expressions
  args <- enquos(...)
  # inline into call
  order_call <- expr(order(!!!args, na.last = !!.na.last))
  # evaluate, raising error if order is somehow the wrong length
  ord <- eval_tidy(order_call, .df)
  stopifnot(length(ord) == nrow(.df))
  # reorder
  .df[ord, , drop = FALSE]
}
```

Using Tidy Evaluation

1.

The use of `$` with an unquoted symbol is harder to understand but avoids the need to convert the symbol to a string.

Base Evaluation

1.

In the caller environment, `data` is the name of a function, not a data frame; it only designates a data frame in the execution environment. Unquoting `data` would fix this.

2.

I take the factory function approach.

```
auto_lm <- function(resp, .data) {
  resp <- ensym(resp)
  .data <- ensym(.data)
  formula <-
    expr(!!resp ~ !!terms)
  lm_call <- expr(lm(!!formula, data = !!.data))
  new_function(exprs(terms = ), expr({
    terms <- enexpr(terms)
    lm_call <- expr(lm(!!resp ~ !!quote(!!terms), data = !!.data))
    eval(lm_call)
  }))
}

test <- auto_lm(mpg, mtcars)
test(disp * cyl)
```

```
Call:
lm(formula = mpg ~ disp * cyl, data = mtcars)

Coefficients:
(Intercept)         disp          cyl
   49.03721     -0.14553     -3.40524
   disp:cyl
    0.01585
```

3.

This results in some kludgy code but evades the problem of `data` not existing in the caller environment.

```
resample <- function(df, n) {
  idx <- sample(nrow(df), n, replace = TRUE)
```

```r
  df[idx, , drop = FALSE]
}

resample_lm2 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  resample_data <- resample(data, n = nrow(data))

  lm_env <- env(env, resample_data = resample_data)
  lm_call <- expr(lm(!!formula, data = data[sample(nrow(data),
    replace = TRUE
  ), , drop = FALSE]))
  expr_print(lm_call)
  eval(lm_call)
}
df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10), 2))
resample_lm2(y ~ x, data = df)
```

```
lm(y ~ x, data = data[sample(nrow(data), replace = TRUE),
, drop = FALSE])
```

```
Call:
lm(formula = y ~ x, data = data[sample(nrow(data), replace = TRUE),
    , drop = FALSE])

Coefficients:
(Intercept)            x
      5.868        2.896
```