Chapter 18: Expressions

Advanced R Book Group, Cohort 3

Megan Stodel (@MeganStodel, www.meganstodel.com)

25 January, 2020 & 1 February, 2021

Expressions allow us to separate our description of the action from the action itself!

```
num_days <- num_weeks * 7
```

```
## Error in eval(expr, envir, enclos): object 'num_weeks' not found
```

This doesn't work because `num_weeks` doesn't exist...yet. But we can still describe this action using `expr()`.

```
library(rlang)
```

```
## Warning: package 'rlang' was built under R version 4.0.3
```

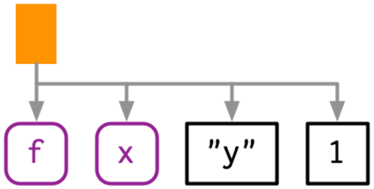
```
day_count <- rlang::expr(num_days <- num_weeks * 7)
num_weeks <- 4
eval(day_count)

# Did it work?
num_days
```

```
## [1] 28
```

Abstract Syntax Trees (AST)

Drawing ASTs

Option 1: Drawn out in a visual diagram



Option 2: With `lobstr::ast()`

```
lobstr::ast(f(x, "y", 1))
#> █─f
#> ├─x
#> ├─"y"
#> └─1
```

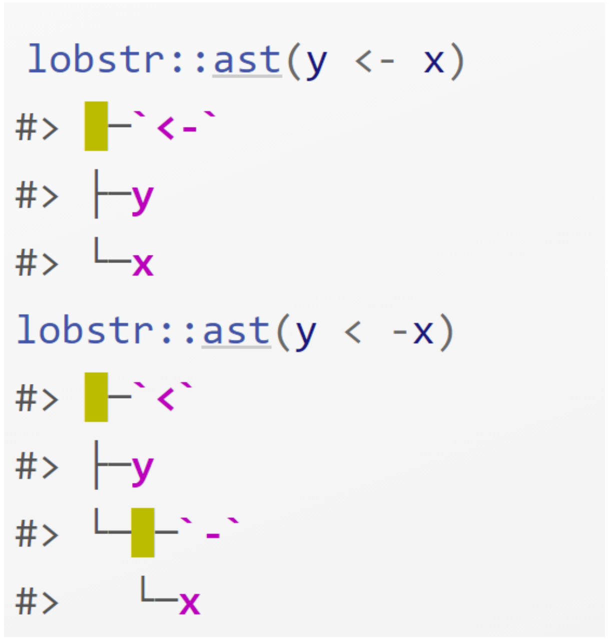
Function calls are drawn as orange rectangles. Symbols are in purple with rounded corners. Constants are black with squared corners.

The first child is the function called (`f` in this example)

Non-code components

Called 'Abstract' because don't capture details unimportant to structure, like comments or whitespace.

There's only one example of whitespace being relevant:

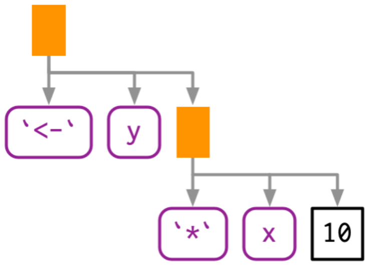


```
lobstr::ast(y <- x)
#> █─`<-`
#> ├─y
#> └─x
lobstr::ast(y < -x)
#> █─`<`
#> ├─y
#> └─█─`-`
#>   └─x
```

Infix calls

Infix calls, like `<-` and `+` can be written in prefix form, so therefore can be written in tree form.

```r
y <- x * 10

# is the same as...
`<-`(y, `*`(x, 10))
```



Generating an expression with prefix calls will lead to it being printed in infix form if relevant.

```r
rlang::expr(`<-`(y, `*`(x, 10)))
```

```
## y <- x * 10
```

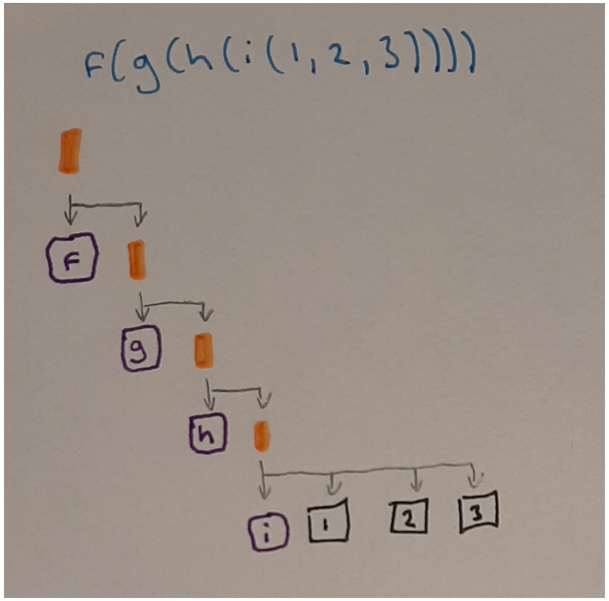
18.2.4, Exercise 1

(This looked like it wasn't displaying right for me so I skipped…)

18.2.4, Exercise 2

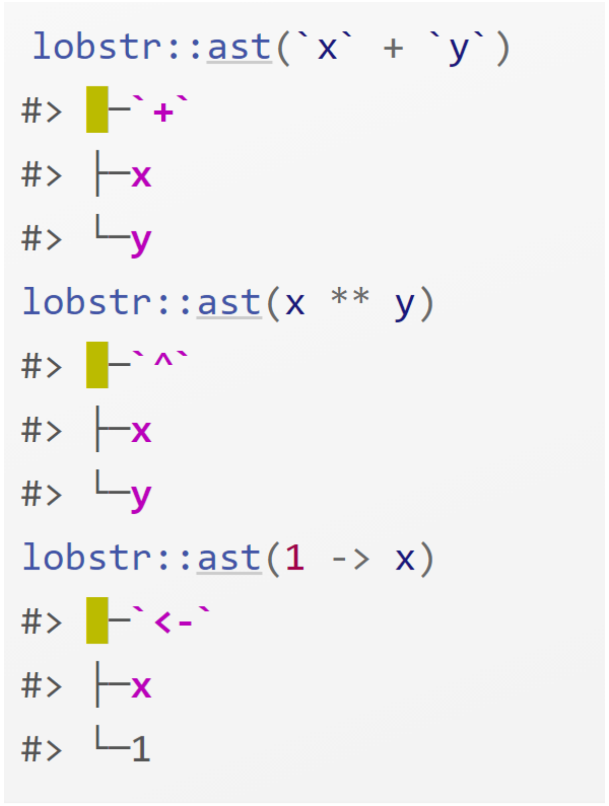
Draw the following trees by hand and then check your answers with lobstr::ast().

a) `f(g(h(i(1, 2, 3))))`



18.2.4, Exercise 3

What's happening with the ASTs below? (Hint: carefully read `?"^"`.)



```
lobstr::ast(`x` + `y`)
#> █─`+`
#> ├─x
#> └─y
lobstr::ast(x ** y)
#> █─`^`
#> ├─x
#> └─y
lobstr::ast(1 -> x)
#> █─`<-`
#> ├─x
#> └─1
```

18.2.4, Exercise 4

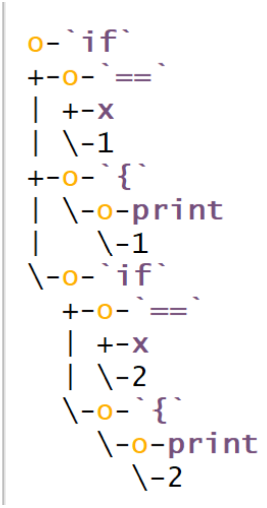
What is special about the AST below? (Hint: re-read Section 6.2.1.)

```
lobstr::ast(function(x = 1, y = 2) {})
#> █─`function`
#> ├─█─x = 1
#> │ └─y = 2
#> ├─█─`{`
#> └─<inline srcref>
```

18.2.4, Exercise 5

What does the call tree of an if statement with multiple else if conditions look like? Why?

```r
lobstr::ast(if (x == 1) {
  print(1)
} else if (x == 2) {
  print(2)
}
)
```



```
o-`if`
+-o-`==`
| +-x
| \-1
+-o-`{`
| \-o-print
|   \-1
\-o-`if`
  +-o-`==`
  | +-x
  | \-2
  \-o-`{`
    \-o-print
      \-2
```

Expressions (yes there is a section of the chapter Expressions called Expressions)

An expression is any member of the set of base types created by parsing code: constant scalars, symbols, call objects, and pairlists.

Constants

A constant is either `NULL` or a length-1 atomic vector like `TRUE`, `1L`, `2.5` or `"x"`. You can test for a constant with `rlang::is_syntactic_literal()`.

These are self-quoting: the expression used to represent a constant is the same constant.

```
identical(rlang::expr(2L), 2L)
```

```
## [1] TRUE
```

```
identical(rlang::expr("advanced_r"), "advanced_r")
```

```
## [1] TRUE
```

Symbols

A symbol represents the name of an object. A symbol is always length 1.

There are two ways to create a symbol:

```r
# Use expr() to capture code referencing an object
rlang::expr(x)
```

```
## x
```

```r
# Use rlang::sym() to turn string into a symbol
rlang::sym("x")
```

```
## x
```

```r
# Turn back into a string
rlang::as_string(rlang::expr(x))
```

```
## [1] "x"
```

Calls

A call object represents a captured function call. Use `is.call()` to check because `typeof()` and `str()` are weird.

```
x <- rlang::expr(data.table::fread("important.csv",
                                   blank.lines.skip=TRUE))

typeof(x)
```

```
## [1] "language"
```

```
is.call(x)
```

```
## [1] TRUE
```

You can use normal subsetting tools with calls because they are like lists.

```
# First element is function
x[[1]]
```

```
## data.table::fread
```

Subsetting with arguments (1)

```
as.list(x[-1])
```

```
## [[1]]
## [1] "important.csv"
##
## $blank.lines.skip
## [1] TRUE
```

```
# Can also extract named arguments
x$blank.lines.skip
```

```
## [1] TRUE
```

```
# Number of arguments (minus 1 because of function element)
length(x) - 1
```

```
## [1] 2
```

Subsetting with arguments (2)

R is very flexible with argument matching - can be in any location and use full, abbreviated or no name. So use `call_standardise()` for consistency.

```
rlang::call_standardise(x)
```

```
## data.table::fread(input = "important.csv", blank.lines.skip = TRUE)
```

And modify like any list.

```
x$header <- TRUE
x
```

```
## data.table::fread("important.csv", blank.lines.skip = TRUE, header = TRUE)
```

Constructing calls

```r
rlang::call2("mean",
             x = rlang::expr(x),
             na.rm = TRUE)
```

```
## mean(x = x, na.rm = TRUE)
```

```r
# First arg is name of function to call,
# remaining arguments will be passed to the call
```

18.3.5, Exercise 1

Which two of the six types of atomic vector can't appear in an expression? Why? Similarly, why can't you create an expression that contains an atomic vector of length greater than one?

Raw and Complex atomics are created using functions. Same with an atomic vector of length greater than 1 (`c()`).

Expressions that include functions are calls.

18.3.5, Exercise 2

What happens when you subset a call object to remove the first element, e.g. `expr(read.csv("foo.csv", header = TRUE))[-1]`. Why?

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

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

It treats the first argument as a function because the function is always the first element.

18.3.5, Exercise 3

Describe the differences between the following call objects.

```r
x <- 1:10

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

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

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

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

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

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

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

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

18.3.5, Exercise 4

`rlang::call_standardise()` doesn't work so well for the following calls. Why? What makes `mean()` special?

```r
rlang::call_standardise(quote(mean(1:10, na.rm = TRUE)))
```

```
## mean(x = 1:10, na.rm = TRUE)
```

```r
rlang::call_standardise(quote(mean(n = T, 1:10)))
```

```
## mean(x = 1:10, n = T)
```

```r
rlang::call_standardise(quote(mean(x = 1:10, , TRUE)))
```

```
## mean(x = 1:10, , TRUE)
```

Because `mean()` uses `...` to access all its arguments, it can't have its argument standardised.

18.3.5, Exercise 5

Why does this code not make sense?

```
x <- rlang::expr(foo(x = 1))
names(x) <- c("x", "")
```

This gives a name to the first element but that is weird because the first element is always a function.

```
x
```

```
## foo(1)
```

So it hasn't done anything except remove the first argument's name.

18.3.5, Exercise 6

Construct the expression `if(x > 1) "a" else "b"` using multiple calls to `call2()`. How does the code structure reflect the structure of the AST?

```r
rlang::call2("if", rlang::call2(">", rlang::sym("x"), 1), "a", "b")
```

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

Parsing and Grammar

The process by which a computer language takes a string and constructs an expression is called parsing, and is governed by a set of rules known as a grammar.

Operator precedence

There can be ambiguity with infix functions, so there are conventions over what takes precedence.

This is fairly straightforward with arithmetic because it uses the same conventions as outside programming!

More complicated with others…there are over 30 operators in 18 precedence groups.

Can find details in `?Syntax` but can also make sure things are happening in the order you want by using brackets.

Associativity

What is the precedence when the same infix operator is used twice? Normally doesn't matter BUT some S3 classes define + in a non-associative way.

E.g. ggplot2 overloads + to build up a complex plot from simple pieces; this is non-associative because earlier layers are drawn underneath later layers.

Most operators are therefore left-associative - operations on the left are evaluated first.

Two exceptions are exponentiation and assignment.

Parsing

If you have code stored in a string, and you want to parse it yourself, you can do so using `rlang::parse_expr()`.

```r
x1 <- "y <- x + 10"
x1
```

```
## [1] "y <- x + 10"
```

```r
is.call(x1)
```

```
## [1] FALSE
```

```r
x2 <- rlang::parse_expr(x1)
x2
```

```
## y <- x + 10
```

```r
is.call(x2)
```

```
## [1] TRUE
```

18.4.4, Exercise 1

R uses parentheses in two slightly different ways as illustrated by these two calls:

```r
f((1))
`(`(1 + 1)
```

Compare and contrast the two uses by referencing the AST. (can be a part of R's general prefix function syntax but can also represent a call to the (function.

```r
lobstr::ast(f((1)))
```

```
## o-f
## \-o-`(`
##   \-1
```

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

```
## o-`(`
## \-o-`+`
##   +-1
```

18.4.4, Exercise 2

= can also be used in two ways. Construct a simple example that shows both uses.

```
example_equals = runif(n = 100)
```

18.4.4, Exercise 3

Does `-2^2` yield 4 or -4? Why?

Check operator precedence in `?Syntax` - `^` has a higher precedence so -4.

```
-2^2
```

```
## [1] -4
```

18.4.4, Exercise 4

What does `!1 + !1` return? Why?

```
!1 + !1
```

```
## [1] FALSE
```

1) The right `!1` evaluates to FALSE. 1 is converted to TRUE, as a logical operator is applied. The negation of TRUE then equals FALSE.

2) 1 + FALSE is evaluated to 1, since FALSE is coerced to 0.

3) Finally !1 is evaluated to FALSE.

18.4.4, Exercise 5

Why does `x1 <- x2 <- x3 <- 0` work? Describe the two reasons. 1) `<-` is right-associative, i.e. evaluation takes place from right to left. 2) `<-` invisibly returns the value on the right-hand side.

18.4.4, Exercise 6

Compare the ASTs of `x + y %+% z` and `x ^ y %+% z`. What have you learned about the precedence of custom infix functions?

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

```
## o-`+`
## +-x
## \-o-`%+%`
##   +-y
##   \-z
```

```
lobstr::ast(x ^ y %+% z)
```

```
## o-`%+%`
## +-o-`^`
## | +-x
## | \-y
## \-z
```

`%+%` has precedence between `+` and `^`.

18.4.4, Exercise 7 onwards...

This seemed like too many exercises.

Walking AST with recursive functions

codetools

The base codetools package provides two interesting functions:

`findGlobals()` locates all global variables used by a function. This can be useful if you want to check that your function doesn't inadvertently rely on variables defined in their parent environment.

`checkUsage()` checks for a range of common problems including unused local variables, unused parameters, and the use of partial argument matching.

Recursive functions

Recursive functions are a natural fit to tree-like data structures because a recursive function is made up of two parts that correspond to the two parts of the tree.

The recursive case handles the nodes in the tree. Typically, you'll do something to each child of a node, usually calling the recursive function again, and then combine the results back together again. For expressions, you'll need to handle calls and pairlists (function arguments).

The base case handles the leaves of the tree. The base cases ensure that the function eventually terminates, by solving the simplest cases directly. For expressions, you need to handle symbols and constants in the base case.

Helper functions

```r
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 {
    typeof(x)
  }
}

expr_type(rlang::expr("a"))
```

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

```r
expr_type(rlang::expr(x))
```

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

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

```r
recurse_call <- function(x) {
  switch_expr(x,
    # Base cases
    symbol = ,
    constant = ,

    # Recursive cases
    call = ,
    pairlist =
  )
}
```

Finding F and T

We'll start with a function that determines whether another function uses the logical abbreviations T and F because using them is often considered to be poor coding practice. Our goal is to return TRUE if the input contains a logical abbreviation, and FALSE otherwise.

```
# Check types
expr_type(rlang::expr(TRUE))
```

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

```
expr_type(rlang::expr(T))
```

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

Create F and T finding function

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    constant = FALSE,
```

Finding all variables created by assignment

List all variables created by assignment.

We start by implementing the base cases and providing a helpful wrapper around the recursive function. Here the base cases are straightforward because we know that neither a symbol nor a constant represents assignment.

```r
find_assign_rec <- function(x) {
  switch_expr(x,
    constant = ,
    symbol = character()
  )
}
find_assign <- function(x) find_assign_rec(enexpr(x))

find_assign("x")
```

```
## character(0)
```

```r
find_assign(x)
```

```
## character(0)
```

Recursive cases

`flat_map_chr()` expects .f to return a character vector of arbitrary length, and flattens all results into a single character vector.

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

flat_map_chr(letters[1:3], ~ rep(., sample(3, 1)))
```

```
## [1] "a" "b" "c"
```

Recursive cases

The recursive case for pairlists is straightforward: we iterate over every element of the pairlist (i.e. each function argument) and combine the results. The case for calls is a little bit more complex: if this is a call to `<-` then we should return the second element of the call.

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

    # Recursive cases
    pairlist = flat_map_chr(as.list(x), find_assign_rec),
    call = {
      if (is_call(x, "<-")) {
        as_string(x[[2]])
      } else {
        flat_map_chr(as.list(x), find_assign_rec)
      }
    }
  )
}
```

Assigning the same variable multiple times?

```
find_assign({
  a <- 1
  a <- 2
})
```

```
## [1] "a" "a"
```

Fix:

```
find_assign <- function(x) unique(find_assign_rec(enexpr(x)))
find_assign({
  a <- 1
  a <- 2
})
```

```
## [1] "a"
```

Nested calls to <-?

Immediately terminate recursion at first `<-`.

```
find_assign({
  a <- b <- c <- 1
})
```

```
## [1] "a"
```

Extract out `find_assign_call()` into a separate function.

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

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

find_assign_rec <- function(x) {
```

18.5.3, Exercise 1

`logical_abbr()` returns `TRUE` for `T(1, 2, 3)`. How could you modify `logical_abbr_rec()` so that it ignores function calls that use `T` or `F`?

```
find_T_call <- function(x) {
  if (is_call(x, "T")) {
    x <- as.list(x)[-1]
    purrr::some(x, logical_abbr_rec)
  } else {
    purrr::some(x, logical_abbr_rec)
  }
}

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

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

18.5.3, Exercise 2

`logical_abbr()` works with expressions. It currently fails when you give it a function. Why? How could you modify `logical_abbr()` to make it work? What components of a function will you need to recurse over?

`switch_expr()` does not handle closure.

18.5.3, Exercise 3

Ran out of time for more exercises…

Specialised data structures

Pairlists

The only place you are likely to see pairlists in R is when working with calls to the `function` function, as the formal arguments to a function are stored in a pairlist.

You can just treat like a list.

Missing arguments

The empty symbol is used to represent missing arguments. You only need to care about the missing symbol if you're programmatically creating functions with missing arguments.

You can make an empty symbol with `missing_arg()` (or `expr()`). An empty symbol doesn't print anything, so you can check if you have one with `rlang::is_missing()`.

This is particularly important for `...` which is always associated with an empty symbol.

```
f <- rlang::expr(function(...) list(...))
args <- f[[2]]
rlang::is_missing(args[[1]])
```

```
## [1] TRUE
```

Expression vectors

Expression vectors are only produced by two base functions: `expression()` and `parse()`.

They behave like lists.