Expressions with Rlang

Kirsten

25 sep 2018

Install needed packages

```
devtools::install_github("r-lib/rlang")
devtools::install_githu("hadley/lobstr")
# Rtools version 3.4 are needed for lobstr
require("rlang","lobstr")
```

Introduction I

Save expressitions not containg values

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

```
## Error in eval(expr, envir, enclos): objekt 'x' blev ikke
```

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

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

So we see, that `expr()` add quotationmarks. To evaluate the expression for a value of x, we can write

Introduction II

```
x <- 10
l <- eval(z)
l
```

```
## [1] 100
```

Notice that `eval()`is a scilent function so we need to assign the output to a value.

AST

We have just created what we call a **quoted expression**. These are what we call **abstract syntax trees** (AST).

Illustration on blackboard for a function `f(x,"y",1)`.

In AST we define.

- **Calls**, define the hierarchy of the tree. This is mainly relevant when we consider multiple functions.
- **The children**, which are ordered as following: **The first child** are the function, **the subsequent children** are the arguments.

As we have learned earlier the order of the children are important.

AST II

- **The leaves** (another name for the children), are **symbols**, that is objects without a value, such as `f`and `x` or **constants** such as `1` or `"Y"`.

AST III

Show the AST tree with `lobstr::ast()`

```r
lobstr::ast(f(x,"y",1))
```

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

Q: Run this code in R

Unquoting

We can unquote an expression using the function `!!`. Try to Write the following in R, What would you expect, what do you see?

```
x <- expr(f(x,"y",1))

lobstr::ast(x)
lobstr::ast(!!x)
```

Unquoting II

```
x <- rlang::expr(f(x,"y",1))

lobstr::ast(x)
```

```
## x
```

```
lobstr::ast(!!x)
```

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

Infix and prefix

Try to run the following code.

```
View(expr(y <- x*10))
```

You will get a detailed describtion, containing, the type of the children, the evaluation order, and the functions. You can even go through the AST tree by unfolding the different calls.

Infix and prefix

- **Infix form** is when a function come inbetween arguments, such as `<-` and `*` above.
- **Prefix functions** are functions where the arguments come first.

An infix function can be turned into a prefix function

```
x <- 4
y <- x * 10
y
```

```
## [1] 40
```

```
`<-`(z, `*`(x, 10))
z
```

```
## [1] 40
```

Draw the AST on the blackboard.

Exercise

Print the AST for `names(x)<-y`.

- Which calls to you make?
- What type are the children, and what is the order of the children?

Try to run `lobstr::ast(mean(x = mtcars$cyl, na.rm = TRUE))` (you might need to type `data(mtcars)` first.).

- Can you find a infix and a prefix function.
- Is the output as expected?

Other special forms that we can look at are `for()`, `if()`, [, [[,{ and many more.

Grammar

The rules that governer the trees constructed by a sequence of tokens, are called the **grammar**.

This includes the order of which operations ar processed, forexample do R read `(1+2)*3` or `1+(2*3)`, if we write `1+2*3`.

Q: What would you expect?

- Write `lobstr::ast(1 + 2 * 3)`to find out.
- What if we write `lobstr::ast(!x %in% y)`?

If you write `lobstr::ast(1 + 2 + 3)` you see, that R is **left-associative**, meaning values on the left are evaluated first. However the exceptions that confirms the rule are `<-` and `^`. We can sometimes overrule the left associative. To see this try to consider `x + y %+% z` and `x ^ y %+% z`.

Data structure

Expressions are used to refer to the set containing **constants, symbols, pairlists** and **calls**.

In **base R** expressions (called **expressions object** in [HW]) is a special type equivalent to a list of expressions.

The expressions we will be working with are closest to the R base object called **langauge object** (including symbols and calls). The type you saw ealier when using `view()`.

Constants

Constants occure in the leaves of AST, and are the simplest datastructure found in the AST (that is they are **atomic vectors** of length 1).

Constants are self quoting since

```
identical(expr("x"), "x")
```

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

```
identical(expr(TRUE), TRUE)
```

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

Q: do you think the same goes for `identical(expr(1), 1)`? Try to type it.

Symbols I

Symbols represent variable names. We can convert back and fourth between symbols and the string that represents them.

```
"x"
```

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

```
sym("x")
```

```
## x
```

```
as_string(sym("x"))
```

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

Symbols II

We can also use syms to put the symbols in a list.

```
syms(c("h","hey"))
```

```
## [[1]]
## h
##
## [[2]]
## hey
```

Symbols III

Symbols differ from string mainly in evaluation.

```r
x <- 10
y <- "5"

z<- eval(expr(f <- x*4))
z
```

```
## [1] 40
```

```r
y <- eval(expr(f <- y*4))
```

```
## Error in y * 4: non-numeric argument to binary operator
```

Missing arguments

A special symbol is the missing arguments.

```r
rlang::missing_arg()
```

```r
rlang::type_of(missing_arg())
```

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

```r
rlang::as_string(missing_arg())
```

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

Notice that the type of `missing_arg()` is a symbol, but if we turn it into a string it contains no value.

Mising arguments II

To see if a symbol is missing, we can use

```
rlang::is_missing(missing_arg())
```

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

We can bind it to a variable `m <- missing_arg()`, what would you expect the output to be?

Missing arguments III

```
m <- missing_arg()
m
```

```
## Error in eval(expr, envir, enclos): argument "m" is miss
```

If we instead write

```
mm <- list(missing_arg())
mm[[1]]
```

We will not get an error or an output at all.

To be able to work with missing arguments stored in a variable we can write,

```
rlang::maybe_missing(m)
```

Calls

Calls define the AST, and behave similar to a list.

```r
x <- expr(read.table("important.csv", row = FALSE))

ast(!!x)
```

```
## o-read.table
## +-"important.csv"
## \-row = FALSE
```

Calls II

```r
length(x) -1 # Get the number of arguments
```

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

```r
names(x) # Missing arguments
```

```
## [1] ""    ""    "row"
```

```r
c(x[[1]],x[[2]]) # Extract leaves
```

```
## [[1]]
## read.table
##
## [[2]]
## [1] "important.csv"
```

```r
x$row
```

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

Extract arguments I

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

```
## read.table(file = "important.csv", row.names = FALSE)
```

`call_standardise()` standardises all arguments to use the full name. However if the function uses `...` it is not possible to standardise all arguments.

Extract arguments II

Now we can extract multiple arguments

```r
as.list(x[2:3])
```

```
## [[1]]
## [1] "important.csv"
##
## $row
## [1] FALSE
```

It is possible to use without `as.list()`, but it is considererd good pratice.

Modify calls

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

```
## read.table("important.csv", row = FALSE, header = TRUE)
```

Construct calls

Finaly, we can construct a call from its children using `call2()`

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

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

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

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

```
x <- 1:10
eval(call2(expr(mean), x = expr(x), na.rm = TRUE))
```

```
## [1] 5.5
```

Q: try to construct a call using your favorite function.

Pairlists

Pairlists have mostly been replaced by lists, but will occure when working with function arguments.

```r
f <- function(x = 10) x + 1
typeof(formals(f))
```

```
## [1] "pairlist"
```

The disadvantage is that the datastructure is a linked list instrad of a vector, so subsetting gets slower the further down the pairlist you index.

Pairlists II

In C the pairlists are still used. But we can treat them as a list.

```
pl <- pairlist(x = 1, y = 2)
length(pl)
```

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

```
str(pl)
```

```
## Dotted pair list of 2
##  $ x: num 1
##  $ y: num 2
```

To illustrate the disadvantage

```
l1 <- as.list(1:100)
l2 <- as.pairlist(1:100)
```

Pairlist III

`supressWarnings()` not to print the warning that the computation time of `l1[[1]]`and `l1[[100]]` are to short to measure

```
suppressWarnings(microbenchmark::microbenchmark(
  l1[[1]],
  l1[[100]],
  l2[[1]],
  l2[[100]]))
```

```
## Unit: nanoseconds
##        expr min  lq   mean median   uq  max neval
##     l1[[1]]   0   0  98.95      0    0 9168   100
##   l1[[100]]   0   0   0.36      0    1    1   100
##     l2[[1]] 353 706 801.10    706  706 8111   100
##   l2[[100]] 705 706 875.06    706 1058 3174   100
```

Expression objects.

Remember these are a equivalent to a list of expressions.

We can produce an expression object in the two following ways in base R.

```
exp1 <- parse(text = c("
x <- 4
x
"))
exp2 <- expression(x <- 4,x)

typeof(exp1)
```

```
## [1] "expression"
```

```
typeof(exp2)
```

```
## [1] "expression"
```

Expression objects II

Q: Try to write `length(exp1)` and `exp1[[1]]`.

If we use `eval()` on a expression object, we evaluate all expressions and thus get a list of expressions.

Parsing and deparsing

How can we evaluate an expression if it is stored as a string, lets say

```
x <- "y <- z + 7"
lobstr::ast(!!x)
```

```
## "y <- z + 7"
```

Parsing I

Using `rlang()` we can do the following

```
x1 <- rlang::parse_expr(x)
x1
```

```
## y <- z + 7
```

```
# With AST
lobstr::ast(!!x1)
```

```
## o-`<-`
## +-y
## \-o-`+`
##   +-z
##   \-7
```

Parsing II

For multiple expressions in a string

```
x <- "a <- 1; a + 1"
rlang::parse_exprs(x)
```

Q: What would you expect?

Parsing III

```r
x <- "a <- 1; a + 1"
rlang::parse_exprs(x)
```

```
## [[1]]
## a <- 1
##
## [[2]]
## a + 1
```

The basse R function equivalent to `parse_exprs()` is `parse()`.

Deparsing I

```r
z <- expr(y <- x + 10)
expr_text(z)
```

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

With R base equivalent function being `deparse()`.

Quasiquotation

Quasiquotation is one of the fundamental ideas that make the functions `expr()` and `ast()` work.

- **quotation** allows us to capture AST associated with an argument.
- **Unquotation** allows you to selectively evaluate parts of an quoted expression.

Quotation I

Why do we need unquoting?

```r
paste("It", "takes", "a","while","to","type","all","these",
```

```
## [1] "It takes a while to type all these words"
```

exprs() I

The function exprs() makes this easier

```
cement <- function(...) {
  dots <- exprs(...)
  paste(purrr::map(dots, expr_name), collapse = " ")
}

cement(now,I,can,write,the,words,without,quotation)
```

```
## [1] "now I can write the words without quotation"
```

exprs() II

However it doesnt solve all problems

```
day <- "Friday"

paste("Today is",day)
```

```
## [1] "Today is Friday"
```

```
cement("Today is",day)
```

```
## [1] "Today is day"
```

Q: Can you remember a function that might help us?

Unquoting: Introduction

To solve this problem we need a function to unqoute.

```
cement("Today is",!!day)
```

```
## [1] "Today is Friday"
```

With rlang

4 functions,

- `expr()` and `enexpr()`
- `exprs()` and `enexprs()`

We have already used `expr()`.

```r
expr(1 /2 / 3)
```

```
## 1/2/3
```

Exercise

What would you expect the output from the following code to be?

```
f1 <- function(x) expr(x)
f1(a + b + c)
```

Answer

```r
f1 <- function(x) expr(x)
f1(a + b + c)
```

```
## x
```

enexpr()

But with enexpr()

```r
f1 <- function(x) enexpr(x)
f1(a + b + c)
```

```
## a + b + c
```

expres()

With exprs() we can evaluate a list

```r
exprs(x = x ^ 2, y = y ^ 3, z = z ^ 4)
```

```
## $x
## x^2
##
## $y
## y^3
##
## $z
## z^4
```

`expres()` II

The function also returns missing arguments

```
val <- exprs(x = )
is_missing(val$x)
```

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

In conclusion use è`nexpr()` and `enexprs()` inside functions and `expr()` and `exprs()` to capture expressions.

Exercise

What does teh following code return, and why?

```
expr({
  x +              y # comment
})
```

Answer

```r
expr({
  x +              y # comment
})
```

```
## {
##     x + y
## }
```

Unquoting: rlang

To unquote with rlang we will use the function `!!`.

We have seen before

```
x <- expr(a + b + c)
expr(f(!!x, y))
```

```
## f(a + b + c, y)
```

Diagram of AST?

Unquoting: rlang II

What happens if we write

```
x <- exprs(1, 2, 3, y = 10)
expr(f(!!x, z = z))
```

```
## f(list(1, 2, 3, y = 10), z = z)
```

```
expr(f(!!!x, z = z))
```

```
## f(1, 2, 3, y = 10, z = z)
```

!!! takes a list of expressions and insert them at the location of `!!!`.

Problems with !!!

```r
x <- 100
head(mtcars$cyl,7)
```

```
## [1] 6 6 4 6 8 6 8
```

```r
head(with(mtcars, cyl + !!x),7)
```

```
## [1] 7 7 5 7 9 7 9
```

```r
head(with(mtcars, cyl + !x),7)
```

```
## [1] 6 6 4 6 8 6 8
```

Q: What woud you expect to get if we wrote `!!!x` instea?

Additional

`rlang::eval_bare()` is another relevant function if you want to evaluate an expression.