

# Package ‘lazyeval’

October 29, 2017

**Version** 0.2.1

**Title** Lazy (Non-Standard) Evaluation

**Description** An alternative approach to non-standard evaluation using formulas. Provides a full implementation of LISP style 'quasiquote', making it easier to generate code with other code.

**License** GPL-3

**LazyData** true

**Depends** R (>= 3.1.0)

**Suggests** knitr, rmarkdown (>= 0.2.65), testthat, covr

**VignetteBuilder** knitr

**RoxygenNote** 6.0.1

**NeedsCompilation** yes

**Author** Hadley Wickham [aut, cre],  
RStudio [cph]

**Maintainer** Hadley Wickham <hadley@rstudio.com>

**Repository** CRAN

**Date/Publication** 2017-10-29 22:59:36 UTC

## R topics documented:

as.lazy . . . . .	2
ast_ . . . . .	3
as_name . . . . .	3
call_modify . . . . .	4
call_new . . . . .	5
expr_label . . . . .	5
function_new . . . . .	6
f_capture . . . . .	7
f_eval_rhs . . . . .	8
f_interp . . . . .	9
f_list . . . . .	10

f_new	11
f_rhs	12
f_text	13
f_unwrap	13
interp	14
is_formula	15
is_lang	15
lazy_	16
lazy_dots	17
lazy_eval	18
make_call	19
missing_arg	20

<b>Index</b>	<b>21</b>
--------------	-----------

---

as.lazy	<i>Convert an object to a lazy expression or lazy dots.</i>
---------	---

---

## Description

Convert an object to a lazy expression or lazy dots.

## Usage

```
as.lazy(x, env = baseenv())
```

```
as.lazy_dots(x, env)
```

## Arguments

x	An R object. Current methods for <code>as.lazy()</code> convert formulas, character vectors, calls and names. Methods for <code>as.lazy_dots()</code> convert lists and character vectors (by calling <code>lapply()</code> with <code>as.lazy()</code> .)
env	Environment to use for objects that don't already have associated environment.

## Examples

```
as.lazy(~ x + 1)
as.lazy(quote(x + 1), globalenv())
as.lazy("x + 1", globalenv())

as.lazy_dots(list(~x, y = ~z + 1))
as.lazy_dots(c("a", "b", "c"), globalenv())
as.lazy_dots(~x)
as.lazy_dots(quote(x), globalenv())
as.lazy_dots(quote(f()), globalenv())
as.lazy_dots(lazy(x))
```

---

ast\_ *Display a call (or expression) as a tree.*

---

### Description

ast\_ takes a quoted expression; ast does the quoting for you.

### Usage

```
ast_(x, width = getOption("width"))
```

```
ast(x)
```

### Arguments

x	Quoted call, list of calls, or expression to display.
width	Display width, defaults to current width as reported by getOption("width").

### Examples

```
ast(f(x, 1, g(), h(i())))
ast(if (TRUE) 3 else 4)
ast(function(a = 1, b = 2) {a + b + 10})
ast(f(x)(y)(z))
```

```
ast_(quote(f(x, 1, g(), h(i()))))
ast_(quote(if (TRUE) 3 else 4))
ast_(expression(1, 2, 3))
```

---

as\_name *Coerce an object to a name or call.*

---

### Description

These are a S3 generics with built-in methods for names, calls, formuals, and strings. The distinction between a name and a call is particularly important when coercing from a string. Coercing to a call will parse the string, coercing to a name will create a (potentially) non-syntactic name.

### Usage

```
as_name(x)
```

```
as_call(x)
```

### Arguments

x	An object to coerce
---	---------------------

**Examples**

```
as_name("x + y")
as_call("x + y")

as_call(~ f)
as_name(~ f())
```

---

call_modify	<i>Modify the arguments of a call.</i>
-------------	--

---

**Description**

Modify the arguments of a call.

**Usage**

```
call_modify(call, new_args, env = parent.frame())

call_standardise(call, env = parent.frame())
```

**Arguments**

call	A call to modify. It is first standardised with <a href="#">call_standardise</a> .
new_args	A named list of expressions (constants, names or calls) used to modify the call. Use NULL to remove arguments.
env	Environment in which to look up call value.

**Examples**

```
call <- quote(mean(x, na.rm = TRUE))
call_standardise(call)

# Modify an existing argument
call_modify(call, list(na.rm = FALSE))
call_modify(call, list(x = quote(y)))

# Remove an argument
call_modify(call, list(na.rm = NULL))

# Add a new argument
call_modify(call, list(trim = 0.1))

# Add an explicit missing argument
call_modify(call, list(na.rm = quote(expr = )))
```

---

call_new	<i>Create a call by "hand"</i>
----------	--------------------------------

---

**Description**

Create a call by "hand"

**Usage**

```
call_new(f, ..., .args = list())
```

**Arguments**

f	Function to call. For <code>make_call</code> , either a string, a symbol or a quoted call. For <code>do_call</code> , a bare function name or call.
..., .args	Arguments to the call either in or out of a list

**Examples**

```
# f can either be a string, a symbol or a call
call_new("f", a = 1)
call_new(quote(f), a = 1)
call_new(quote(f()), a = 1)

#' Can supply arguments individually or in a list
call_new(quote(f), a = 1, b = 2)
call_new(quote(f), .args = list(a = 1, b = 2))
```

---

expr_label	<i>Find the expression associated with an argument</i>
------------	--

---

**Description**

`expr_find()` finds the full expression; `expr_text()` turns the expression into a single string; `expr_label()` formats it nicely for use in messages. `expr_env()` finds the environment associated with the expression.

**Usage**

```
expr_label(x)

expr_text(x, width = 60L, nlines = Inf)

expr_find(x)

expr_env(x, default_env)
```

**Arguments**

x	A promise (function argument)
width	Width of each line
nlines	Maximum number of lines to extract.
default_env	If supplied, <code>expr_env</code> will return this if the promise has already been forced. Otherwise it will throw an error.

**Details**

These functions never force promises, and will work even if a promise has previously been forced.

**Examples**

```
# Unlike substitute(), expr_find() finds the original expression
f <- function(x) g(x)
g <- function(y) h(y)
h <- function(z) list(substitute(z), expr_find(z))

f(1 + 2 + 3)

expr_label(10)
# Names a quoted with ``
expr_label(x)
# Strings are encoded
expr_label("a\nb")
# Expressions are captured
expr_label(a + b + c)
# Long expressions are collapsed
expr_label(foo({
  1 + 2
  print(x)
}))
```

---

function\_new

*Create a function by "hand"*


---

**Description**

This constructs a new function given it's three components: list of arguments, body code and parent environment.

**Usage**

```
function_new(args, body, env = parent.frame())
```

**Arguments**

args	A named list of default arguments. Note that if you want arguments that don't have defaults, you'll need to use the special function <code>alist</code> , e.g. <code>alist(a = , b = 1)</code>
body	A language object representing the code inside the function. Usually this will be most easily generated with <code>quote</code>
env	The parent environment of the function, defaults to the calling environment of <code>make_function</code>

**Examples**

```
f <- function(x) x + 3
g <- function_new(alist(x = ), quote(x + 3))

# The components of the functions are identical
identical(formals(f), formals(g))
identical(body(f), body(g))
identical(environment(f), environment(g))

# But the functions are not identical because f has src code reference
identical(f, g)

attr(f, "srcref") <- NULL
# Now they are:
stopifnot(identical(f, g))
```

---

f\_capture

*Make a promise explicit by converting into a formula.*


---

**Description**

This should be used sparingly if you want to implement true non-standard evaluation with 100% magic. I recommend avoiding this unless you have strong reasons otherwise since requiring arguments to be formulas only adds one extra character to the inputs, and otherwise makes life much simpler.

**Usage**

```
f_capture(x)

dots_capture(..., .ignore_empty = TRUE)
```

**Arguments**

x, ...	An unevaluated promises
.ignore_empty	If TRUE, empty arguments will be silently dropped.

**Value**

f\_capture returns a formula; dots\_capture returns a list of formulas.

**Examples**

```
f_capture(a + b)
dots_capture(a + b, c + d, e + f)

# These functions will follow a chain of promises back to the
# original definition
f <- function(x) g(x)
g <- function(y) h(y)
h <- function(z) f_capture(z)
f(a + b + c)
```

---

f\_eval\_rhs

*Evaluate a formula*


---

**Description**

f\_eval\_rhs evaluates the RHS of a formula and f\_eval\_lhs evaluates the LHS. f\_eval is a short-cut for f\_eval\_rhs since that is what you most commonly need.

**Usage**

```
f_eval_rhs(f, data = NULL)
```

```
f_eval_lhs(f, data = NULL)
```

```
f_eval(f, data = NULL)
```

```
find_data(x)
```

**Arguments**

f A formula. Any expressions wrapped in `uq()` will be "unquoted", i.e. they will be evaluated, and the results inserted back into the formula. See [f\\_interp](#) for more details.

data A list (or data frame). `find_data` is a generic used to find the data associated with a given object. If you want to make `f_eval` work for your own objects, you can define a method for this generic.

x An object for which you want to find associated data.

**Details**

If data is specified, variables will be looked for first in this object, and if not found in the environment of the formula.



## Pronouns

When used with data, `f_eval` provides two pronouns to make it possible to be explicit about where you want values to come from: `.env` and `.data`. These are thin wrappers around `.data` and `.env` that throw errors if you try to access non-existent values.

## Examples

```
f_eval(~ 1 + 2 + 3)

# formulas automatically capture their enclosing environment
foo <- function(x) {
  y <- 10
  ~ x + y
}
f <- foo(1)
f
f_eval(f)

# If you supply data, f_eval will look their first:
f_eval(~ cyl, mtcars)

# To avoid ambiguity, you can use .env and .data pronouns to be
# explicit:
cyl <- 10
f_eval(~ .data$cyl, mtcars)
f_eval(~ .env$cyl, mtcars)

# Imagine you are computing the mean of a variable:
f_eval(~ mean(cyl), mtcars)
# How can you change the variable that's being computed?
# The easiest way is "unquote" with uq()
# See ?f_interp for more details
var <- ~ cyl
f_eval(~ mean( uq(var) ), mtcars)
```

---

f\_interp

*Interpolate a formula*

---

## Description

Interpolation replaces sub-expressions of the form `uq(x)` with the evaluated value of `x`, and inlines sub-expressions of the form `uqs(x)`.

## Usage

```
f_interp(f, data = NULL)
```

```
uq(x, data = NULL)
```

```
uqf(x)
```

```
uqs(x)
```

### Arguments

f	A one-sided formula.
data	When called from inside <code>f_eval</code> , this is used to pass on the data so that nested formulas are evaluated in the correct environment.
x	For <code>uq</code> and <code>uqf</code> , a formula. For <code>uqs</code> , a vector.

### Theory

Formally, `f_interp` is a quasiquote function, `uq()` is the unquote operator, and `uqs()` is the unquote splice operator. These terms have a rich history in LISP, and live on in modern languages like <http://docs.julialang.org/en/release-0.1/manual/metaprogramming/> and <https://docs.racket-lang.org/reference/quasiquote.htm>

### Examples

```
f_interp(x ~ 1 + uq(1 + 2 + 3) + 10)

# Use uqs() if you want to add multiple arguments to a function
# It must evaluate to a list
args <- list(1:10, na.rm = TRUE)
f_interp(~ mean( uqs(args) ))

# You can combine the two
var <- quote(xyz)
extra_args <- list(trim = 0.9)
f_interp(~ mean( uq(var) , uqs(extra_args) ))

foo <- function(n) {
  ~ 1 + uq(n)
}
f <- foo(10)
f
f_interp(f)
```

---

f\_list

*Build a named list from the LHS of formulas*

---

### Description

`f_list` makes a new list; `as_f_list` takes an existing list. Both take the LHS of any two-sided formulas and evaluate it, replacing the current name with the result.

**Usage**

```
f_list(...)  
as_f_list(x)
```

**Arguments**

<code>...</code>	Named arguments.
<code>x</code>	An existing list

**Value**

A named list.

**Examples**

```
f_list("y" ~ x)  
f_list(a = "y" ~ a, ~ b, c = ~c)
```

---

<code>f_new</code>	<i>Create a formula object by "hand".</i>
--------------------	---

---

**Description**

Create a formula object by "hand".

**Usage**

```
f_new(rhs, lhs = NULL, env = parent.frame())
```

**Arguments**

<code>lhs, rhs</code>	A call, name, or atomic vector.
<code>env</code>	An environment

**Value**

A formula object

**Examples**

```
f_new(quote(a))  
f_new(quote(a), quote(b))
```

---

`f_rhs`*Get/set formula components.*

---

**Description**

`f_rhs` extracts the righthand side, `f_lhs` extracts the lefthand side, and `f_env` extracts the environment. All functions throw an error if `f` is not a formula.

**Usage**

```
f_rhs(f)
```

```
f_rhs(x) <- value
```

```
f_lhs(f)
```

```
f_lhs(x) <- value
```

```
f_env(f)
```

```
f_env(x) <- value
```

**Arguments**

`f`, `x`            A formula

`value`            The value to replace with.

**Value**

`f_rhs` and `f_lhs` return language objects (i.e. atomic vectors of length 1, a name, or a call). `f_env` returns an environment.

**Examples**

```
f_rhs(~ 1 + 2 + 3)
```

```
f_rhs(~ x)
```

```
f_rhs(~ "A")
```

```
f_rhs(1 ~ 2)
```

```
f_lhs(~ y)
```

```
f_lhs(x ~ y)
```

```
f_env(~ x)
```

---

f_text	<i>Turn RHS of formula into a string/label.</i>
--------	---

---

**Description**

Equivalent of `expr_text()` and `expr_label()` for formulas.

**Usage**

```
f_text(x, width = 60L, nlines = Inf)
```

```
f_label(x)
```

**Arguments**

x	A formula.
width	Width of each line
nlines	Maximum number of lines to extract.

**Examples**

```
f <- ~ a + b + bc
f_text(f)
f_label(f)

# Names a quoted with ``
f_label(~ x)
# Strings are encoded
f_label(~ "a\nb")
# Long expressions are collapsed
f_label(~ foo({
  1 + 2
  print(x)
}))
```

---

f_unwrap	<i>Unwrap a formula</i>
----------	-------------------------

---

**Description**

This interpolates values in the formula that are defined in its environment, replacing the environment with its parent.

**Usage**

```
f_unwrap(f)
```

**Arguments**

f                    A formula to unwrap.

**Examples**

```
n <- 100
f <- ~ x + n
f_unwrap(f)
```

---

interp                    *Interpolate values into an expression.*

---

**Description**

This is useful if you want to build an expression up from a mixture of constants and variables.

**Usage**

```
interp(`_obj`, ..., .values)
```

**Arguments**

\_obj                    An object to modify: can be a call, name, formula, [lazy](#), or a string.  
 ..., .values            Either individual name-value pairs, or a list (or environment) of values.

**Examples**

```
# Interp works with formulas, lazy objects, quoted calls and strings
interp(~ x + y, x = 10)
interp(lazy(x + y), x = 10)
interp(quote(x + y), x = 10)
interp("x + y", x = 10)
```

```
# Use as.name if you have a character string that gives a
# variable name
interp(~ mean(var), var = as.name("mpg"))
# or supply the quoted name directly
interp(~ mean(var), var = quote(mpg))
```

```
# Or a function!
interp(~ f(a, b), f = as.name("+"))
# Remember every action in R is a function call:
# http://adv-r.had.co.nz/Functions.html#all-calls
```

```
# If you've built up a list of values through some other
# mechanism, use .values
interp(~ x + y, .values = list(x = 10))
```

```
# You can also interpolate variables defined in the current
```

```
# environment, but this is a little risky.  
y <- 10  
interp(~ x + y, .values = environment())
```

---

is_formula	<i>Is object a formula?</i>
------------	-----------------------------

---

### Description

Is object a formula?

### Usage

```
is_formula(x)
```

### Arguments

x                    Object to test

### Examples

```
is_formula(~ 10)  
is_formula(10)
```

---

is_lang	<i>Is an object a language object?</i>
---------	--

---

### Description

These helpers are consistent wrappers around their base R equivalents. A language object is either an atomic vector (typically a scalar), a name (aka a symbol), a call, or a pairlist (used for function arguments).

### Usage

```
is_lang(x)
```

```
is_name(x)
```

```
is_call(x)
```

```
is_pairlist(x)
```

```
is_atomic(x)
```

**Arguments**

x                    An object to test.

**See Also**

[as\\_name\(\)](#) and [as\\_call\(\)](#) for coercion functions.

**Examples**

```
q1 <- quote(1)
is_lang(q1)
is_atomic(q1)

q2 <- quote(x)
is_lang(q2)
is_name(q2)

q3 <- quote(x + 1)
is_lang(q3)
is_call(q3)
```

---

lazy\_

*Capture expression for later lazy evaluation.*

---

**Description**

lazy() uses non-standard evaluation to turn promises into lazy objects; lazy\_() does standard evaluation and is suitable for programming.

**Usage**

```
lazy_(expr, env)
```

```
lazy(expr, env = parent.frame(), .follow_symbols = TRUE)
```

**Arguments**

expr                Expression to capture. For lazy\_ must be a name or a call.

env                 Environment in which to evaluate expr.

.follow\_symbols

If TRUE, the default, follows promises across function calls. See vignette("chained-promises") for details.

**Details**

Use lazy() like you'd use [substitute\(\)](#) to capture an unevaluated promise. Compared to substitute() it also captures the environment associated with the promise, so that you can correctly replay it in the future.



**Examples**

```

lazy_(quote(a + x), globalenv())

# Lazy is designed to be used inside a function - you should
# give it the name of a function argument (a promise)
f <- function(x = b - a) {
  lazy(x)
}
f()
f(a + b / c)

# Lazy also works when called from the global environment. This makes
# easy to play with interactively.
lazy(a + b / c)

# By default, lazy will climb all the way back to the initial promise
# This is handy if you have if you have nested functions:
g <- function(y) f(y)
h <- function(z) g(z)
f(a + b)
g(a + b)
h(a + b)

# To avoid this behaviour, set .follow_symbols = FALSE
# See vignette("chained-promises") for details

```

---

lazy\_dots

*Capture ... (dots) for later lazy evaluation.*


---

**Description**

Capture ... (dots) for later lazy evaluation.

**Usage**

```
lazy_dots(..., .follow_symbols = FALSE, .ignore_empty = FALSE)
```

**Arguments**

...	Dots from another function
.follow_symbols	If TRUE, the default, follows promises across function calls. See vignette("chained-promises") for details.
.ignore_empty	If TRUE, empty arguments will be ignored.

**Value**

A named list of [lazy](#) expressions.

**Examples**

```

lazy_dots(x = 1)
lazy_dots(a, b, c * 4)

f <- function(x = a + b, ...) {
  lazy_dots(x = x, y = a + b, ...)
}
f(z = a + b)
f(z = a + b, .follow_symbols = TRUE)

# .follow_symbols is off by default because it causes problems
# with lazy loaded objects
lazy_dots(letters)
lazy_dots(letters, .follow_symbols = TRUE)

# You can also modify a dots like a list. Anything on the RHS will
# be coerced to a lazy.
l <- lazy_dots(x = 1)
l$y <- quote(f)
l[c("y", "x")]
l["z"] <- list(~g)

c(lazy_dots(x = 1), lazy_dots(f))

```

---

lazy\_eval

*Evaluate a lazy expression.*


---

**Description**

Evaluate a lazy expression.

**Usage**

```
lazy_eval(x, data = NULL)
```

**Arguments**

x	A lazy object or a formula.
data	Option, a data frame or list in which to preferentially look for variables before using the environment associated with the lazy object.

**Examples**

```

f <- function(x) {
  z <- 100
  ~ x + z
}
z <- 10
lazy_eval(f(10))

```

```
lazy_eval(f(10), list(x = 100))
lazy_eval(f(10), list(x = 1, z = 1))

lazy_eval(lazy_dots(a = x, b = z), list(x = 10))
```

---

`make_call`*Make a call with lazy\_dots as arguments.*

---

## Description

In order to exactly replay the original call, the environment must be the same for all of the dots. This function circumvents that a little, falling back to the `baseenv()` if all environments aren't the same.

## Usage

```
make_call(fun, args)
```

## Arguments

<code>fun</code>	Function as symbol or quoted call.
<code>args</code>	Arguments to function; must be a <code>lazy_dots</code> object, or something <code>as.lazy_dots()</code> can coerce..

## Value

A list:

<code>env</code>	The common environment for all elements
<code>expr</code>	The expression

## Examples

```
make_call(quote(f), lazy_dots(x = 1, 2))
make_call(quote(f), list(x = 1, y = ~x))
make_call(quote(f), ~x)

# If no known or no common environment, fails back to baseenv()
make_call(quote(f), quote(x))
```

---

missing_arg	<i>Generate a missing argument.</i>
-------------	-------------------------------------

---

**Description**

Generate a missing argument.

**Usage**

```
missing_arg()
```

**Examples**

```
f_interp(~f(x = uq(missing_arg())))  
f_interp(~f(x = uq(NULL)))
```

# Index

alist, 7  
as.lazy, 2  
as.lazy\_dots, 19  
as.lazy\_dots (as.lazy), 2  
as\_call, 16  
as\_call (as\_name), 3  
as\_f\_list (f\_list), 10  
as\_name, 3, 16  
ast (ast\_), 3  
ast\_, 3  
  
baseenv, 19  
  
call\_modify, 4  
call\_new, 5  
call\_standardise, 4  
call\_standardise (call\_modify), 4  
  
dots\_capture (f\_capture), 7  
  
expr\_env (expr\_label), 5  
expr\_find (expr\_label), 5  
expr\_label, 5, 13  
expr\_text, 13  
expr\_text (expr\_label), 5  
  
f\_capture, 7  
f\_env (f\_rhs), 12  
f\_env<- (f\_rhs), 12  
f\_eval (f\_eval\_rhs), 8  
f\_eval\_lhs (f\_eval\_rhs), 8  
f\_eval\_rhs, 8  
f\_interp, 8, 9  
f\_label (f\_text), 13  
f\_lhs (f\_rhs), 12  
f\_lhs<- (f\_rhs), 12  
f\_list, 10  
f\_new, 11  
f\_rhs, 12  
f\_rhs<- (f\_rhs), 12  
f\_text, 13  
  
f\_unwrap, 13  
find\_data (f\_eval\_rhs), 8  
function\_new, 6  
  
interp, 14  
is\_atomic (is\_lang), 15  
is\_call (is\_lang), 15  
is\_formula, 15  
is\_lang, 15  
is\_name (is\_lang), 15  
is\_pairlist (is\_lang), 15  
  
lapply, 2  
lazy, 14, 17  
lazy (lazy\_), 16  
lazy\_, 16  
lazy\_dots, 17  
lazy\_eval, 18  
  
make\_call, 19  
missing\_arg, 20  
  
quote, 7  
  
substitute, 16  
  
uq (f\_interp), 9  
uqf (f\_interp), 9  
uqs (f\_interp), 9