Package 'rlang'

August 9, 2017

Version 0.1.2

Title Functions for Base Types and Core R and 'Tidyverse' Features

Description A toolbox for working with base types, core R features like the condition system, and core 'Tidyverse' features like tidy evaluation.

License GPL-3

LazyData true

Depends R (>= 3.1.0)

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

VignetteBuilder knitr

RoxygenNote 6.0.1

URL http://rlang.tidyverse.org, https://github.com/tidyverse/rlang

BugReports https://github.com/tidyverse/rlang/issues

NeedsCompilation yes

Author Lionel Henry [aut, cre],
Hadley Wickham [aut],
RStudio [cph]

Maintainer Lionel Henry <lionel@rstudio.com>

Repository CRAN

Date/Publication 2017-08-09 20:37:03 UTC

R topics documented:

abort 4
are_na 5
arg_match 6
as_bytes 7
as_env 7
as_function 8
as_overscope 9

as_pairlist . . . 11
as_quosure . . . 11
as_utf8_character . . . 12
bare-type-predicates . . . 13
caller_env . . . 15
call_inspect . . . 15
cnd_signal . . . 16
dictionary . . . 18
dots_definitions . . . 19
dots_list . . . 20
dots_n . . . 22
dots_values . . . 22
duplicate . . . 23
empty_env . . . 23
env . . . 24
env_bind . . . 26
env_bury . . . 29
env_clone . . . 30
env_depth . . . 30
env_get . . . 31
env_has . . . 32
env_inherits . . . 32
env_names . . . 33
env_parent . . . 34
env_unbind . . . 35
eval_bare . . . 36
eval_tidy . . . 37
eval_tidy_ . . . 39
exiting . . . 40
expr . . . 41
exprs_auto_name . . . 42
expr_interp . . . 43
expr_label . . . 44
flatten . . . 45
fn_env . . . 47
fn_fmls . . . 48
frame_position . . . 49
friendly_type . . . 50
f_rhs . . . 50
f_text . . . 51
get_env . . . 52
has_length . . . 53
has_name . . . 54
invoke . . . 55
is_callable . . . 56
is_condition . . . 57
is_copyable . . . 57
is_empty . . . 58

is_env . . . 58
is_expr . . . 59
is_formula . . . 61
is_frame . . . 62
is_function . . . 63
is_installed . . . 64
is_integerish . . . 65
is_lang . . . 66
is_named . . . 67
is_pairlist . . . 68
is_quosure . . . 69
is_stack . . . 70
is_symbol . . . 70
is_true . . . 71
lang . . . 71
lang_args . . . 73
lang_fn . . . 74
lang_head . . . 74
lang_modify . . . 75
lang_name . . . 76
lang_standardise . . . 77
missing . . . 78
missing_arg . . . 79
modify . . . 80
mut_utf8_locale . . . 81
names2 . . . 81
new_cnd . . . 82
new_formula . . . 83
new_function . . . 84
ns_env . . . 85
op-definition . . . 85
op-get-attr . . . 86
op-na-default . . . 87
op-null-default . . . 87
pairlist . . . 88
parse_expr . . . 90
prepend . . . 92
prim_name . . . 92
quasiquotation . . . 93
quo-predicates . . . 94
quosure . . . 95
quo_expr . . . 99
restarting . . . 100
return_from . . . 101
rst_abort . . . 102
rst_list . . . 103
rst_muffle . . . 104
scalar-type-predicates . . . 106

scoped_env . . . 107
seq2 . . . 108
set_attrs . . . 109
set_chr_encoding . . . 110
set_expr . . . 111
set_names . . . 112
splice . . . 113
stack . . . 114
stack_trim . . . 117
string . . . 118
switch_lang . . . 119
switch_type . . . 120
sym . . . 122
tidyeval-data . . . 122
type-predicates . . . 123
type_of . . . 124
vector-along . . . 125
vector-coercion . . . 126
vector-construction . . . 128
vector-len . . . 130
with_env . . . 131
with_handlers . . . 132
with_restarts . . . 133

Index **137**

`abort` *Signal an error, warning, or message*

Description

These functions are equivalent to base functions `base::stop()`, `base::warning()` and `base::message()`, but the `type` argument makes it easy to create subclassed conditions. They also don't include call information by default. This saves you from typing `call. = FALSE` to make error messages cleaner within package functions.

Usage

```
abort(msg, type = NULL, call = FALSE)

warn(msg, type = NULL, call = FALSE)

inform(msg, type = NULL, call = FALSE)
```

Arguments

`msg`	A message to display.
`type`	Subclass of the condition to signal.
`call`	Whether to display the call.

Details

Like `stop()` and `cnd_abort()`, `abort()` signals a critical condition and interrupts execution by jumping to top level (see `rst_abort()`). Only a handler of the relevant type can prevent this jump by making another jump to a different target on the stack (see `with_handlers()`).

`warn()` and `inform()` both have the side effect of displaying a message. These messages will not be displayed if a handler transfers control. Transfer can be achieved by establishing an exiting handler that transfers control to `with_handlers()`). In this case, the current function stops and execution resumes at the point where handlers were established.

Since it is often desirable to continue normally after a message or warning, both `warn()` and `inform()` (and their base R equivalent) establish a muffle restart where handlers can jump to prevent the message from being displayed. Execution resumes normally after that. See `rst_muffle()` to jump to a muffling restart, and the `muffle` argument of `inplace()` for creating a muffling handler.

`are_na` *Test for missing values*

Description

`are_na()` checks for missing values in a vector and is equivalent to `base::is.na()`. It is a vectorised predicate, meaning that its output is always the same length as its input. On the other hand, `is_na()` is a scalar predicate and always returns a scalar boolean, `TRUE` or `FALSE`. If its input is not scalar, it returns `FALSE`. Finally, there are typed versions that check for particular missing types.

Usage

```
are_na(x)

is_na(x)

is_lgl_na(x)

is_int_na(x)

is_dbl_na(x)

is_chr_na(x)

is_cpl_na(x)
```

Arguments

`x`	An object to test

Details

The scalar predicates accept non-vector inputs. They are equivalent to `is_null()` in that respect. In contrast the vectorised predicate `are_na()` requires a vector input since it is defined over vector values.

Examples

```
# are_na() is vectorised and works regardless of the type
are_na(c(1, 2, NA))
are_na(c(1L, NA, 3L))

# is_na() checks for scalar input and works for all types
is_na(NA)
is_na(na_dbl)
is_na(character(0))

# There are typed versions as well:
is_lgl_na(NA)
is_lgl_na(na_dbl)
```

---

arg_match *Match an argument to a character vector*

Description

This is equivalent to `base::match.arg()` with a few differences:

- Partial matches trigger an error.
- Error messages are a bit more informative and obey the tidyverse standards.

Usage

```
arg_match(arg, values = NULL)
```

Arguments

`arg`	A symbol referring to an argument accepting strings.
`values`	The possible values that `arg` can take. If `NULL`, the values are taken from the function definition of the caller frame.

Value

The string supplied to `arg`.

Examples

```
fn <- function(x = c("foo", "bar")) arg_match(x)
fn("bar")

# This would throw an informative error if run:
# fn("b")
# fn("baz")
```

as_bytes *Coerce to a raw vector*

Description

This currently only works with strings, and returns its hexadecimal representation.

Usage

```
as_bytes(x)
```

Arguments

x	A string.

Value

A raw vector of bytes.

as_env *Coerce to an environment*

Description

`as_env()` coerces named vectors (including lists) to an environment. It first checks that `x` is a dictionary (see `is_dictionaryish()`). If supplied an unnamed string, it returns the corresponding package environment (see `pkg_env()`).

Usage

```
as_env(x, parent = NULL)
```

Arguments

x	An object to coerce.
parent	A parent environment, `empty_env()` by default. This argument is only used when `x` is data actually coerced to an environment (as opposed to data representing an environment, like `NULL` representing the empty environment).

Details

If `x` is an environment and `parent` is not `NULL`, the environment is duplicated before being set a new parent. The return value is therefore a different environment than `x`.

Examples

```
# Coerce a named vector to an environment:
env <- as_env(mtcars)

# By default it gets the empty environment as parent:
identical(env_parent(env), empty_env())


# With strings it is a handy shortcut for pkg_env():
as_env("base")
as_env("rlang")

# With NULL it returns the empty environment:
as_env(NULL)
```

---

as_function *Convert to function or closure*

Description

- `as_function()` transform objects to functions. It fetches functions by name if supplied a string or transforms quosures to a proper function.
- `as_closure()` first passes its argument to `as_function()`. If the result is a primitive function, it regularises it to a proper closure (see `is_function()` about primitive functions).

Usage

```
as_function(x, env = caller_env())

as_closure(x, env = caller_env())
```

Arguments

`x`	A function or formula. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs.
`env`	Environment in which to fetch the function in case `x` is a string.

Examples

```
f <- as_function(~ . + 1)
f(10)

# Primitive functions are regularised as closures
as_closure(list)
```

```
as_closure("list")

# Operators have `.x` and `.y` as arguments, just like lambda
# functions created with the formula syntax:
as_closure(`+`)
as_closure(`~`)
```

`as_overscope` *Create a dynamic scope for tidy evaluation*

Description

Tidy evaluation works by rescoping a set of symbols (column names of a data frame for example) to custom bindings. While doing this, it is important to keep the original environment of captured expressions in scope. The gist of tidy evaluation is to create a dynamic scope containing custom bindings that should have precedence when expressions are evaluated, and chain this scope (set of linked environments) to the lexical enclosure of formulas under evaluation. During tidy evaluation, formulas are transformed into formula-promises and will self-evaluate their RHS as soon as they are called. The main trick of tidyeval is to consistently rechain the dynamic scope to the lexical enclosure of each tidy quote under evaluation.

Usage

```
as_overscope(quo, data = NULL)

new_overscope(bottom, top = NULL, enclosure = base_env())

overscope_eval_next(overscope, quo, env = base_env())

overscope_clean(overscope)
```

Arguments

`quo`	A quosure.
`data`	Additional data to put in scope.
`bottom`	This is the environment (or the bottom of a set of environments) containing definitions for overscoped symbols. The bottom environment typically contains pronouns (like `.data`) while its direct parents contain the overscoping bindings. The last one of these parents is the `top`.
`top`	The top environment of the overscope. During tidy evaluation, this environment is chained and rechained to lexical enclosures of self-evaluating formulas (or quosures). This is the mechanism that ensures hygienic scoping: the bindings in the overscope have precedence, but the bindings in the dynamic environment where the tidy quotes were created in the first place are in scope as well.
`enclosure`	The default enclosure. After a quosure is done self-evaluating, the overscope is rechained to the default enclosure.

overscope	A valid overscope containing bindings for ~, .top_env and _F and whose parents contain overscoped bindings for tidy evaluation.
env	The lexical enclosure in case quo is not a validly scoped quosure. This is the base environment by default.

Details

These functions are useful for embedding the tidy evaluation framework in your own DSLs with your own evaluating function. They let you create a custom dynamic scope. That is, a set of chained environments whose bottom serves as evaluation environment and whose top is rechained to the current lexical enclosure. But most of the time, you can just use eval_tidy_() as it will take care of installing the tidyeval components in your custom dynamic scope.

- as_overscope() is the function that powers eval_tidy(). It could be useful if you cannot use eval_tidy() for some reason, but serves mostly as an example of how to build a dynamic scope for tidy evaluation. In this case, it creates pronouns .data and .env and buries all dynamic bindings from the supplied data in new environments.
- new_overscope() is called by as_overscope() and eval_tidy_(). It installs the definitions for making formulas self-evaluate and for formula-guards. It also installs the pronoun .top_env that helps keeping track of the boundary of the dynamic scope. If you evaluate a tidy quote with eval_tidy_(), you don't need to use this.
- eval_tidy_() is useful when you have several quosures to evaluate in a same dynamic scope. That's a simple wrapper around eval_bare() that updates the .env pronoun and rechains the dynamic scope to the new formula enclosure to evaluate.
- Once an expression has been evaluated in the tidy environment, it's a good idea to clean up the definitions that make self-evaluation of formulas possible overscope_clean(). Otherwise your users may face unexpected results in specific corner cases (e.g. when the evaluation environment is leaked, see examples). Note that this function is automatically called by eval_tidy_().

Value

An overscope environment.

A valid overscope: a child environment of bottom containing the definitions enabling tidy evaluation (self-evaluating quosures, formula-unguarding, ...).

Examples

```
# Evaluating in a tidy evaluation environment enables all tidy
# features:
expr <- quote(list(.data$cyl, ~letters))
f <- as_quosure(expr)
overscope <- as_overscope(f, data = mtcars)
overscope_eval_next(overscope, f)

# However you need to cleanup the environment after evaluation.
# Otherwise the leftover definitions for self-evaluation of
# formulas might cause unexpected results:
fn <- overscope_eval_next(overscope, ~function() ~letters)
```

```
fn()

overscope_clean(overscope)
fn()
```

`as_pairlist` *Coerce to pairlist*

Description

This transforms vector objects to a linked pairlist of nodes. See pairlist for information about the pairlist type.

Usage

```
as_pairlist(x)
```

Arguments

`x`	An object to coerce.

See Also

pairlist

`as_quosure` *Coerce object to quosure*

Description

Quosure objects wrap an expression with a lexical enclosure. This is a powerful quoting (see `base::quote()` and `quo()`) mechanism that makes it possible to carry and manipulate expressions while making sure that its symbolic content (symbols and named calls, see `is_symbolic()`) is correctly looked up during evaluation.

- `new_quosure()` creates a quosure from a raw expression and an environment.
- `as_quosure()` is useful for functions that expect quosures but allow specifying a raw expression as well. It has two possible effects: if `x` is not a quosure, it wraps it into a quosure bundling `env` as scope. If `x` is an unscoped quosure (see `is_quosure()`), `env` is used as a default scope. On the other hand if `x` has a valid enclosure, it is returned as is (even if `env` is not the same as the formula environment).
- While `as_quosure()` always returns a quosure (a one-sided formula), even when its input is a formula or a definition, `as_quosureish()` returns quosureish inputs as is.

Usage

```
as_quosure(x, env = caller_env())

as_quosureish(x, env = caller_env())
```

Arguments

x	An object to convert.
env	An environment specifying the lexical enclosure of the quosure.

See Also

is_quosure()

Examples

```
# Sometimes you get unscoped formulas because of quotation:
f <- ~~expr
inner_f <- f_rhs(f)
str(inner_f)
is_quosureish(inner_f, scoped = TRUE)

# You can use as_quosure() to provide a default environment:
as_quosure(inner_f, base_env())

# Or convert expressions or any R object to a validly scoped quosure:
as_quosure(quote(expr), base_env())
as_quosure(10L, base_env())


# While as_quosure() always returns a quosure (one-sided formula),
# as_quosureish() returns quosureish objects:
as_quosure(a := b)
as_quosureish(a := b)
as_quosureish(10L)
```

---

as_utf8_character *Coerce to a character vector and attempt encoding conversion*

Description

Unlike specifying the encoding argument in as_string() and as_character(), which is only declarative, these functions actually attempt to convert the encoding of their input. There are two possible cases:

- The string is tagged as UTF-8 or latin1, the only two encodings for which R has specific support. In this case, converting to the same encoding is a no-op, and converting to native always works as expected, as long as the native encoding, the one specified by the LC_CTYPE locale (see mut_utf8_locale()) has support for all characters occurring in the strings. Unrepresentable characters are serialised as unicode points: "<U+xxxx>".

- The string is not tagged. R assumes that it is encoded in the native encoding. Conversion to native is a no-op, and conversion to UTF-8 should work as long as the string is actually encoded in the locale codeset.

Usage

```
as_utf8_character(x)

as_native_character(x)

as_utf8_string(x)

as_native_string(x)
```

Arguments

x	An object to coerce.

Examples

```
# Let's create a string marked as UTF-8 (which is guaranteed by the
# Unicode escaping in the string):
utf8 <- "caf\uE9"
str_encoding(utf8)
as_bytes(utf8)

# It can then be converted to a native encoding, that is, the
# encoding specified in the current locale:
## Not run:
mut_latin1_locale()
latin1 <- as_native_string(utf8)
str_encoding(latin1)
as_bytes(latin1)

## End(Not run)
```

---

bare-type-predicates *Bare type predicates*

Description

These predicates check for a given type but only return TRUE for bare R objects. Bare objects have no class attributes. For example, a data frame is a list, but not a bare list.

Usage

```
is_bare_list(x, n = NULL)

is_bare_atomic(x, n = NULL)

is_bare_vector(x, n = NULL)

is_bare_double(x, n = NULL)

is_bare_integer(x, n = NULL)

is_bare_numeric(x, n = NULL)

is_bare_character(x, n = NULL, encoding = NULL)

is_bare_logical(x, n = NULL)

is_bare_raw(x, n = NULL)

is_bare_string(x, n = NULL)

is_bare_bytes(x, n = NULL)
```

Arguments

x	Object to be tested.
n	Expected length of a vector.
encoding	Expected encoding of a string or character vector. One of `UTF-8`, `latin1`, or `unknown`.

Details

- The predicates for vectors include the `n` argument for pattern-matching on the vector length.
- Like `is_atomic()` and unlike base R `is.atomic()`, `is_bare_atomic()` does not return `TRUE` for `NULL`.
- Unlike base R `is.numeric()`, `is_bare_double()` only returns `TRUE` for floating point numbers.

See Also

type-predicates, scalar-type-predicates

caller_env *Get the environment of the caller frame*

Description

`caller_frame()` is a shortcut for `call_frame(2)` and `caller_fn()` and `caller_env()` are shortcuts for `call_frame(2)$env` `call_frame(2)$fn`.

Usage

```
caller_env(n = 1)

caller_frame(n = 1)

caller_fn(n = 1)
```

Arguments

`n`	The number of generation to go back. Note that contrarily to `call_frame()`, 1 represents the parent frame rather than the current frame.

See Also

`call_frame()`

call_inspect *Inspect a call*

Description

This function is useful for quick testing and debugging when you manipulate expressions and calls. It lets you check that a function is called with the right arguments. This can be useful in unit tests for instance. Note that this is just a simple wrapper around `base::match.call()`.

Usage

```
call_inspect(...)
```

Arguments

`...`	Arguments to display in the returned call.

Examples

```
call_inspect(foo(bar), "" %>% identity())
invoke(call_inspect, list(a = mtcars, b = letters))
```

cnd_signal *Signal a condition*

Description

Signal a condition to handlers that have been established on the stack. Conditions signalled with `cnd_signal()` are assumed to be benign. Control flow can resume normally once the conditions has been signalled (if no handler jumped somewhere else on the evaluation stack). On the other hand, `cnd_abort()` treats the condition as critical and will jump out of the distressed call frame (see `rst_abort()`), unless a handler can deal with the condition.

Usage

```
cnd_signal(.cnd, ..., .msg = NULL, .call = FALSE, .mufflable = TRUE)

cnd_abort(.cnd, ..., .msg = NULL, .call = FALSE, .mufflable = FALSE)
```

Arguments

`.cnd`	Either a condition object (see `new_cnd()`), or the name of a s3 class from which a new condition will be created.
`...`	Named data fields stored inside the condition object. These dots are evaluated with explicit splicing.
`.msg`	A string to override the condition's default message.
`.call`	Whether to display the call of the frame in which the condition is signalled. If `TRUE`, the call is stored in the `call` field of the condition object: this field is displayed by R when an error is issued. The call information is also stored in the `.call` field in all cases.
`.mufflable`	Whether to signal the condition with a muffling restart. This is useful to let `inplace()` handlers muffle a condition. It stops the condition from being passed to other handlers when the inplace handler did not jump elsewhere. `TRUE` by default for benign conditions, but `FALSE` for critical ones, since in those cases execution should probably not be allowed to continue normally.

Details

If `.critical` is `FALSE`, this function has no side effects beyond calling handlers. In particular, execution will continue normally after signalling the condition (unless a handler jumped somewhere else via `rst_jump()` or by being `exiting()`). If `.critical` is `TRUE`, the condition is signalled via `base::stop()` and the program will terminate if no handler dealt with the condition by jumping out of the distressed call frame.

`inplace()` handlers are called in turn when they decline to handle the condition by returning normally. However, it is sometimes useful for an inplace handler to produce a side effect (signalling another condition, displaying a message, logging something, etc), prevent the condition from being passed to other handlers, and resume execution from the place where the condition was signalled.

The easiest way to accomplish this is by jumping to a restart point (see `with_restarts()`) established by the signalling function. If `.mufflable` is `TRUE`, a muffle restart is established. This allows inplace handler to muffle a signalled condition. See `rst_muffle()` to jump to a muffling restart, and the `muffle` argument of `inplace()` for creating a muffling handler.

See Also

`abort()`, `warn()` and `inform()` for signalling typical R conditions. See `with_handlers()` for establishing condition handlers.

Examples

```
# Creating a condition of type "foo"
cnd <- new_cnd("foo")

# If no handler capable of dealing with "foo" is established on the
# stack, signalling the condition has no effect:
cnd_signal(cnd)

# To learn more about establishing condition handlers, see
# documentation for with_handlers(), exiting() and inplace():
with_handlers(cnd_signal(cnd),
  foo = inplace(function(c) cat("side effect!\n"))
)


# By default, cnd_signal() creates a muffling restart which allows
# inplace handlers to prevent a condition from being passed on to
# other handlers and to resume execution:
undesirable_handler <- inplace(function(c) cat("please don't call me\n"))
muffling_handler <- inplace(function(c) {
  cat("muffling foo...\n")
  rst_muffle(c)
})

with_handlers(foo = undesirable_handler,
  with_handlers(foo = muffling_handler, {
    cnd_signal("foo")
    "return value"
  }))


# You can signal a critical condition with cnd_abort(). Unlike
# cnd_signal() which has no side effect besides signalling the
# condition, cnd_abort() makes the program terminate with an error
# unless a handler can deal with the condition:
## Not run:
cnd_abort(cnd)

## End(Not run)

# If you don't specify a .msg or .call, the default message/call
```

```
# (supplied to new_cnd()) are displayed. Otherwise, the ones
# supplied to cnd_abort() and cnd_signal() take precedence:
## Not run:
critical <- new_cnd("my_error",
  .msg = "default 'my_error' msg",
  .call = quote(default(call))
)
cnd_abort(critical)
cnd_abort(critical, .msg = "overridden msg")

fn <- function(...) {
  cnd_abort(critical, .call = TRUE)
}
fn(arg = foo(bar))

## End(Not run)

# Note that by default a condition signalled with cnd_abort() does
# not have a muffle restart. That is because in most cases,
# execution should not continue after signalling a critical
# condition.
```

`dictionary` *Create a dictionary*

Description

Dictionaries are a concept of types modelled after R environments. Dictionaries are containers of R objects that:

- Contain uniquely named objects.
- Can only be indexed by name. They must implement the extracting operators `$` and `[[`. The latter returns an error when indexed by position because dictionaries are not vectors (they are unordered).
- Report a clear error message when asked to extract a name that does not exist. This error message can be customised with the `lookup_msg` constructor argument.

Usage

```
as_dictionary(x, lookup_msg = NULL, read_only = FALSE)

is_dictionary(x)
```

Arguments

`x`	An object for which you want to find associated data.
`lookup_msg`	An error message when your data source is accessed inappropriately (by position rather than name).
`read_only`	Whether users can replace elements of the dictionary.

Details

Dictionaries are used within the tidy evaluation framework for creating pronouns that can be explicitly referred to from captured code. See `eval_tidy()`.

`dots_definitions` *Tidy quotation of multiple expressions and dots*

Description

`quos()` quotes its arguments and returns them as a list of quosures (see `quo()`). It is especially useful to capture arguments forwarded through `...`.

Usage

```
dots_definitions(..., .named = FALSE)

quos(..., .named = FALSE, .ignore_empty = c("trailing", "none", "all"))

is_quosures(x)
```

Arguments

`...`	Expressions to capture unevaluated.
`.named`	Whether to ensure all dots are named. Unnamed elements are processed with `expr_text()` to figure out a default name. If an integer, it is passed to the `width` argument of `expr_text()`, if `TRUE`, the default width is used. See `exprs_auto_name()`.
`.ignore_empty`	Whether to ignore empty arguments. Can be one of `"trailing"`, `"none"`, `"all"`. If `"trailing"`, only the last argument is ignored if it is empty.
`x`	An object to test.

Details

Both `quos` and `dots_definitions()` have specific support for definition expressions of the type `var := expr`, with some differences:

`quos()` When `:=` definitions are supplied to `quos()`, they are treated as a synonym of argument assignment `=`. On the other hand, they allow unquoting operators on the left-hand side, which makes it easy to assign names programmatically.

`dots_definitions()` This dots capturing function returns definitions as is. Unquote operators are processed on capture, in both the LHS and the RHS. Unlike `quos()`, it allows named definitions.

Examples

```
# quos() is like the singular version but allows quoting
# several arguments:
quos(foo(), bar(baz), letters[1:2], !! letters[1:2])

# It is most useful when used with dots. This allows quoting
# expressions across different levels of function calls:
fn <- function(...) quos(...)
fn(foo(bar), baz)

# Note that quos() does not check for duplicate named
# arguments:
fn <- function(...) quos(x = x, ...)
fn(x = a + b)


# Dots can be spliced in:
args <- list(x = 1:3, y = ~var)
quos(!!! args, z = 10L)

# Raw expressions are turned to formulas:
args <- alist(x = foo, y = bar)
quos(!!! args)


# Definitions are treated similarly to named arguments:
quos(x := expr, y = expr)

# However, the LHS of definitions can be unquoted. The return value
# must be a symbol or a string:
var <- "foo"
quos(!!var := expr)

# If you need the full LHS expression, use dots_definitions():
dots <- dots_definitions(var = foo(baz) := bar(baz))
dots$defs
```

---

dots_list *Extract dots with splicing semantics*

Description

These functions evaluate all arguments contained in `...` and return them as a list. They both splice their arguments if they qualify for splicing. See `ll()` for information about splicing and below for the kind of arguments that qualify for splicing.

Usage

```
dots_list(..., .ignore_empty = c("trailing", "none", "all"))
```

```
dots_splice(..., .ignore_empty = c("trailing", "none", "all"))
```

Arguments

`...`	Arguments with explicit (`dots_list()`) or list (`dots_splice()`) splicing semantics. The contents of spliced arguments are embedded in the returned list.
`.ignore_empty`	Whether to ignore empty arguments. Can be one of `"trailing"`, `"none"`, `"all"`. If `"trailing"`, only the last argument is ignored if it is empty.

Details

`dots_list()` has *explicit splicing semantics*: it splices lists that are explicitly marked for splicing with the `splice()` adjective. `dots_splice()` on the other hand has *list splicing semantics*: in addition to lists marked explicitly for splicing, bare lists are spliced as well.

Value

A list of arguments. This list is always named: unnamed arguments are named with the empty string `""`.

See Also

`exprs()` for extracting dots without evaluation.

Examples

```
# Compared to simply using list(...) to capture dots, dots_list()
# splices explicitly:
x <- list(1, 2)
dots_list(!!! x, 3)

# Unlike dots_splice(), it doesn't splice bare lists:
dots_list(x, 3)

# Splicing is also helpful to workaround exact and partial matching
# of arguments. Let's create a function taking named arguments and
# dots:
fn <- function(data, ...) {
  dots_list(...)
}

# You normally cannot pass an argument named `data` through the dots
# as it will match `fn`'s `data` argument. The splicing syntax
# provides a workaround:
fn(some_data, !!! list(data = letters))

# dots_splice() splices lists marked with splice() as well as bare
# lists:
x <- list(1, 2)
dots_splice(!!! x, 3)
dots_splice(x, 3)
```

dots_n *How many arguments are currently forwarded in dots?*

Description

This returns the number of arguments currently forwarded in `...` as an integer.

Usage

```
dots_n(...)
```

Arguments

`...`	Forwarded arguments.

Examples

```
fn <- function(...) dots_n(..., baz)
fn(foo, bar)
```

dots_values *Evaluate dots with preliminary splicing*

Description

This is a tool for advanced users. It captures dots, processes unquoting and splicing operators, and evaluates them. Unlike `dots_list()` and `dots_splice()`, it does not flatten spliced objects. They are merely attributed a `spliced` class (see `splice()`). You can process spliced objects manually, perhaps with a custom predicate (see `flatten_if()`).

Usage

```
dots_values(..., .ignore_empty = c("trailing", "none", "all"))
```

Arguments

`...`	Arguments to evaluate and process splicing operators.
`.ignore_empty`	Whether to ignore empty arguments. Can be one of `"trailing"`, `"none"`, `"all"`. If `"trailing"`, only the last argument is ignored if it is empty.

Examples

```
dots <- dots_values(!!! list(1))
dots

# Flatten the spliced objects:
flatten_if(dots, is_spliced)
```

duplicate *Duplicate an R object*

Description

In R semantics, objects are copied by value. This means that modifying the copy leaves the original object intact. Since, copying data in memory is an expensive operation, copies in R are as lazy as possible. They only happen when the new object is actually modified. However, some operations (like `mut_node_car()` or `mut_node_cdr()`) do not support copy-on-write. In those cases, it is necessary to duplicate the object manually in order to preserve copy-by-value semantics.

Usage

```
duplicate(x, shallow = FALSE)
```

Arguments

`x`	Any R object. However, uncopyable types like symbols and environments are returned as is (just like with `<-`).
`shallow`	This is relevant for recursive data structures like lists, calls and pairlists. A shallow copy only duplicates the top-level data structure. The objects contained in the list are still the same.

Details

Some objects are not duplicable, like symbols and environments. `duplicate()` returns its input for these unique objects.

See Also

pairlist

empty_env *Get the empty environment*

Description

The empty environment is the only one that does not have a parent. It is always used as the tail of a scope chain such as the search path (see `scoped_names()`).

Usage

```
empty_env()
```

Examples

```
# Create environments with nothing in scope:
child_env(empty_env())
```

env *Create a new environment*

Description

These functions create new environments.

- `env()` always creates a child of the current environment.
- `child_env()` lets you specify a parent (see section on inheritance).
- `new_environment()` creates a child of the empty environment. It is useful e.g. for using environments as containers of data rather than as part of a scope hierarchy.

Usage

```
env(...)

child_env(.parent, ...)

new_environment(data = list())
```

Arguments

`..., data`	Named values. The dots have explicit splicing semantics.
`.parent`	A parent environment. Can be an object supported by `as_env()`.

Environments as objects

Environments are containers of uniquely named objects. Their most common use is to provide a scope for the evaluation of R expressions. Not all languages have first class environments, i.e. can manipulate scope as regular objects. Reification of scope is one of the most powerful feature of R as it allows you to change what objects a function or expression sees when it is evaluated.

Environments also constitute a data structure in their own right. They are dictionaries of uniquely named objects, subsettable by name and modifiable by reference. This latter property (see section on reference semantics) is especially useful for creating mutable OO systems (cf the R6package and the ggprotosystem for extending ggplot2).

Inheritance

All R environments (except the empty environment) are defined with a parent environment. An environment and its grandparents thus form a linear hierarchy that is the basis for lexicalscoping in R. When R evaluates an expression, it looks up symbols in a given environment. If it cannot find these symbols there, it keeps looking them up in parent environments. This way, objects defined in child environments have precedence over objects defined in parent environments.

The ability of overriding specific definitions is used in the tidyeval framework to create powerful domain-specific grammars. A common use of overscoping is to put data frame columns in scope. See `as_overscope()` for technical details.

Reference semantics

Unlike regular objects such as vectors, environments are an uncopyable object type. This means that if you have multiple references to a given environment (by assigning the environment to another symbol with <- or passing the environment as argument to a function), modifying the bindings of one of those references changes all other references as well.

See Also

scoped_env, env_has(), env_bind().

Examples

```
# env() creates a new environment which has the current environment
# as parent
env <- env(a = 1, b = "foo")
env$b
identical(env_parent(env), get_env())


# child_env() lets you specify a parent:
child <- child_env(env, c = "bar")
identical(env_parent(child), env)

# This child environment owns `c` but inherits `a` and `b` from `env`:
env_has(child, c("a", "b", "c", "d"))
env_has(child, c("a", "b", "c", "d"), inherit = TRUE)

# `parent` is passed to as_env() to provide handy shortcuts. Pass a
# string to create a child of a package environment:
child_env("rlang")
env_parent(child_env("rlang"))

# Or `NULL` to create a child of the empty environment:
child_env(NULL)
env_parent(child_env(NULL))

# The base package environment is often a good default choice for a
# parent environment because it contains all standard base
# functions. Also note that it will never inherit from other loaded
# package environments since R keeps the base package at the tail
# of the search path:
base_child <- child_env("base")
env_has(base_child, c("lapply", "("), inherit = TRUE)

# On the other hand, a child of the empty environment doesn't even
# see a definition for `(`
empty_child <- child_env(NULL)
env_has(empty_child, c("lapply", "("), inherit = TRUE)

# Note that all other package environments inherit from base_env()
# as well:
rlang_child <- child_env("rlang")
```

```
env_has(rlang_child, "env", inherit = TRUE)     # rlang function
env_has(rlang_child, "lapply", inherit = TRUE)  # base function


# Both env() and child_env() take dots with explicit splicing:
objs <- list(b = "foo", c = "bar")
env <- env(a = 1, !!! objs)
env$c

# You can also unquote names with the definition operator `:=`
var <- "a"
env <- env(!!var := "A")
env$a


# Use new_environment() to create containers with the empty
# environment as parent:
env <- new_environment()
env_parent(env)

# Like other new_ constructors, it takes an object rather than dots:
new_environment(list(a = "foo", b = "bar"))
```

`env_bind` *Bind symbols to objects in an environment*

Description

These functions create bindings in an environment. The bindings are supplied through `...` as pairs of names and values or expressions. `env_bind()` is equivalent to evaluating a `<-` expression within the given environment. This function should take care of the majority of use cases but the other variants can be useful for specific problems.

- `env_bind()` takes named *values*. The arguments are evaluated once (with explicit splicing) and bound in `.env`. `env_bind()` is equivalent to `base::assign()`.
- `env_bind_fns()` takes named *functions* and creates active bindings in `.env`. This is equivalent to `base::makeActiveBinding()`. An active binding executes a function each time it is evaluated. `env_bind_fns()` takes dots with implicit splicing, so that you can supply both named functions and named lists of functions.

  If these functions are closures they are lexically scoped in the environment that they bundle. These functions can thus refer to symbols from this enclosure that are not actually in scope in the dynamic environment where the active bindings are invoked. This allows creative solutions to difficult problems (see the implementations of `dplyr::do()` methods for an example).
- `env_bind_exprs()` takes named *expressions*. This is equivalent to `base::delayedAssign()`. The arguments are captured with `exprs()` (and thus support call-splicing and unquoting) and assigned to symbols in `.env`. These expressions are not evaluated immediately but lazily. Once a symbol is evaluated, the corresponding expression is evaluated in turn and its value is bound to the symbol (the expressions are thus evaluated only once, if at all).

Usage

```
env_bind(.env, ...)

env_bind_exprs(.env, ..., .eval_env = caller_env())

env_bind_fns(.env, ...)
```

Arguments

`.env`	An environment or an object bundling an environment, e.g. a formula, quosure or closure. This argument is passed to `get_env()`.
`...`	Pairs of names and expressions, values or functions. These dots support splicing (with varying semantics, see above) and name unquoting.
`.eval_env`	The environment where the expressions will be evaluated when the symbols are forced.

Value

The input object `.env`, with its associated environment modified in place, invisibly.

Side effects

Since environments have reference semantics (see relevant section in `env()` documentation), modifying the bindings of an environment produces effects in all other references to that environment. In other words, `env_bind()` and its variants have side effects.

As they are called primarily for their side effects, these functions follow the convention of returning their input invisibly.

Examples

```
# env_bind() is a programmatic way of assigning values to symbols
# with `<-`. We can add bindings in the current environment:
env_bind(get_env(), foo = "bar")
foo

# Or modify those bindings:
bar <- "bar"
env_bind(get_env(), bar = "BAR")
bar

# It is most useful to change other environments:
my_env <- env()
env_bind(my_env, foo = "foo")
my_env$foo

# A useful feature is to splice lists of named values:
vals <- list(a = 10, b = 20)
env_bind(my_env, !!! vals, c = 30)
my_env$b
my_env$c
```

```
# You can also unquote a variable referring to a symbol or a string
# as binding name:
var <- "baz"
env_bind(my_env, !!var := "BAZ")
my_env$baz


# env_bind() and its variants are generic over formulas, quosures
# and closures. To illustrate this, let's create a closure function
# referring to undefined bindings:
fn <- function() list(a, b)
fn <- set_env(fn, child_env("base"))

# This would fail if run since `a` etc are not defined in the
# enclosure of fn() (a child of the base environment):
# fn()

# Let's define those symbols:
env_bind(fn, a = "a", b = "b")

# fn() now sees the objects:
fn()

# env_bind_exprs() assigns expressions lazily:
env <- env()
env_bind_exprs(env, name = cat("forced!\n"))
env$name
env$name

# You can unquote expressions. Note that quosures are not
# supported, only raw expressions:
expr <- quote(message("forced!"))
env_bind_exprs(env, name = !! expr)
env$name

# You can create active bindings with env_bind_fns()
# Let's create some bindings in the lexical enclosure of `fn`:
counter <- 0

# And now a function that increments the counter and returns a
# string with the count:
fn <- function() {
  counter <<- counter + 1
  paste("my counter:", counter)
}

# Now we create an active binding in a child of the current
# environment:
env <- env()
env_bind_fns(env, symbol = fn)

# `fn` is executed each time `symbol` is evaluated or retrieved:
```

```
env$symbol
env$symbol
eval_bare(quote(symbol), env)
eval_bare(quote(symbol), env)
```

---

env_bury *Overscope bindings by defining symbols deeper in a scope*

Description

env_bury() is like env_bind() but it creates the bindings in a new child environment. This makes sure the new bindings have precedence over old ones, without altering existing environments. Unlike env_bind(), this function does not have side effects and returns a new environment (or object wrapping that environment).

Usage

```
env_bury(.env, ...)
```

Arguments

.env	An environment or an object bundling an environment, e.g. a formula, quosure or closure. This argument is passed to get_env().
...	Pairs of names and expressions, values or functions. These dots support splicing (with varying semantics, see above) and name unquoting.

Value

A copy of .env enclosing the new environment containing bindings to ... arguments.

See Also

env_bind(), env_unbind()

Examples

```
orig_env <- env(a = 10)
fn <- set_env(function() a, orig_env)

# fn() currently sees `a` as the value `10`:
fn()

# env_bury() will bury the current scope of fn() behind a new
# environment:
fn <- env_bury(fn, a = 1000)
fn()

# Even though the symbol `a` is still defined deeper in the scope:
orig_env$a
```

env_clone *Clone an environment*

Description

This creates a new environment containing exactly the same objects, optionally with a new parent.

Usage

```
env_clone(env, parent = env_parent(env))
```

Arguments

env	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
parent	The parent of the cloned environment.

Examples

```
env <- env(!!! mtcars)
clone <- env_clone(env)
identical(env, clone)
identical(env$cyl, clone$cyl)
```

env_depth *Depth of an environment chain*

Description

This function returns the number of environments between env and the empty environment, including env. The depth of env is also the number of parents of env (since the empty environment counts as a parent).

Usage

```
env_depth(env)
```

Arguments

env	An environment or an object bundling an environment, e.g. a formula, quosure or closure.

Value

An integer.

See Also

The section on inheritance in env() documentation.

Examples

```
env_depth(empty_env())
env_depth(pkg_env("rlang"))
```

---

env_get *Get an object from an environment*

Description

env_get() extracts an object from an enviroment env. By default, it does not look in the parent environments.

Usage

```
env_get(env = caller_env(), nm, inherit = FALSE)
```

Arguments

env	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
nm	The name of a binding.
inherit	Whether to look for bindings in the parent environments.

Value

An object if it exists. Otherwise, throws an error.

Examples

```
parent <- child_env(NULL, foo = "foo")
env <- child_env(parent, bar = "bar")

# This throws an error because `foo` is not directly defined in env:
# env_get(env, "foo")

# However `foo` can be fetched in the parent environment:
env_get(env, "foo", inherit = TRUE)
```

env_has *Does an environment have or see bindings?*

Description

`env_has()` is a vectorised predicate that queries whether an environment owns bindings personally (with `inherit` set to `FALSE`, the default), or sees them in its own environment or in any of its parents (with `inherit = TRUE`).

Usage

```
env_has(env = caller_env(), nms, inherit = FALSE)
```

Arguments

`env`	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
`nms`	A character vector containing the names of the bindings to remove.
`inherit`	Whether to look for bindings in the parent environments.

Value

A logical vector as long as `nms`.

Examples

```
parent <- child_env(NULL, foo = "foo")
env <- child_env(parent, bar = "bar")

# env does not own `foo` but sees it in its parent environment:
env_has(env, "foo")
env_has(env, "foo", inherit = TRUE)
```

env_inherits *Does environment inherit from another environment?*

Description

This returns `TRUE` if `x` has `ancestor` among its parents.

Usage

```
env_inherits(env, ancestor)
```

Arguments

env	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
ancestor	Another environment from which x might inherit.

---

env_names *Names of symbols bound in an environment*

Description

env_names() returns object names from an enviroment env as a character vector. All names are returned, even those starting with a dot.

Usage

```
env_names(env)
```

Arguments

env	An environment or an object bundling an environment, e.g. a formula, quosure or closure.

Value

A character vector of object names.

Names of symbols and objects

Technically, objects are bound to symbols rather than strings, since the R interpreter evaluates symbols (see is_expr() for a discussion of symbolic objects versus literal objects). However it is often more convenient to work with strings. In rlang terminology, the string corresponding to a symbol is called the *name* of the symbol (or by extension the name of an object bound to a symbol).

Encoding

There are deep encoding issues when you convert a string to symbol and vice versa. Symbols are *always* in the native encoding (see set_chr_encoding()). If that encoding (let's say latin1) cannot support some characters, these characters are serialised to ASCII. That's why you sometimes see strings looking like <U+1234>, especially if you're running Windows (as R doesn't support UTF-8 as native encoding on that platform).

To alleviate some of the encoding pain, env_names() always returns a UTF-8 character vector (which is fine even on Windows) with unicode points unserialised.

Examples

```
env <- env(a = 1, b = 2)
env_names(env)
```

env_parent *Get parent environments*

Description

- `env_parent()` returns the parent environment of `env` if called with `n = 1`, the grandparent with `n = 2`, etc.
- `env_tail()` searches through the parents and returns the one which has `empty_env()` as parent.
- `env_parents()` returns the list of all parents, including the empty environment.

See the section on *inheritance* in `env()`'s documentation.

Usage

```
env_parent(env = caller_env(), n = 1)

env_tail(env = caller_env())

env_parents(env = caller_env())
```

Arguments

`env`	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
`n`	The number of generations to go up.

Value

An environment for `env_parent()` and `env_tail()`, a list of environments for `env_parents()`.

Examples

```
# Get the parent environment with env_parent():
env_parent(global_env())

# Or the tail environment with env_tail():
env_tail(global_env())

# By default, env_parent() returns the parent environment of the
# current evaluation frame. If called at top-level (the global
# frame), the following two expressions are equivalent:
env_parent()
env_parent(base_env())

# This default is more handy when called within a function. In this
# case, the enclosure environment of the function is returned
# (since it is the parent of the evaluation frame):
```

```
enclos_env <- env()
fn <- set_env(function() env_parent(), enclos_env)
identical(enclos_env, fn())
```

`env_unbind` *Remove bindings from an environment*

Description

`env_unbind()` is the complement of `env_bind()`. Like `env_has()`, it ignores the parent environments of `env` by default. Set `inherit` to `TRUE` to track down bindings in parent environments.

Usage

```
env_unbind(env = caller_env(), nms, inherit = FALSE)
```

Arguments

`env`	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
`nms`	A character vector containing the names of the bindings to remove.
`inherit`	Whether to look for bindings in the parent environments.

Value

The input object `env` with its associated environment modified in place, invisibly.

Examples

```
data <- set_names(as_list(letters), letters)
env_bind(environment(), !!! data)
env_has(environment(), letters)

# env_unbind() removes bindings:
env_unbind(environment(), letters)
env_has(environment(), letters)

# With inherit = TRUE, it removes bindings in parent environments
# as well:
parent <- child_env(NULL, foo = "a")
env <- child_env(parent, foo = "b")
env_unbind(env, "foo", inherit = TRUE)
env_has(env, "foo", inherit = TRUE)
```

`eval_bare` *Evaluate an expression in an environment*

Description

`eval_bare()` is a lightweight version of the base function `base::eval()`. It does not accept supplementary data, but it is more efficient and does not clutter the evaluation stack. Technically, `eval_bare()` is a simple wrapper around the C function `Rf_eval()`.

Usage

```
eval_bare(expr, env = parent.frame())
```

Arguments

`expr`	An expression to evaluate.
`env`	The environment in which to evaluate the expression.

Details

`base::eval()` inserts two call frames in the stack, the second of which features the `envir` parameter as frame environment. This may unnecessarily clutter the evaluation stack and it can change evaluation semantics with stack sensitive functions in the case where `env` is an evaluation environment of a stack frame (see `ctxt_stack()`). Since the base function `eval()` creates a new evaluation context with `env` as frame environment there are actually two contexts with the same evaluation environment on the stack when `expr` is evaluated. Thus, any command that looks up frames on the stack (stack sensitive functions) may find the parasite frame set up by `eval()` rather than the original frame targetted by `env`. As a result, code evaluated with `base::eval()` does not have the property of stack consistency, and stack sensitive functions like `base::return()`, `base::parent.frame()` may return misleading results.

See Also

with_env

Examples

```
# eval_bare() works just like base::eval():
env <- child_env(NULL, foo = "bar")
expr <- quote(foo)
eval_bare(expr, env)

# To explore the consequences of stack inconsistent semantics, let's
# create a function that evaluates `parent.frame()` deep in the call
# stack, in an environment corresponding to a frame in the middle of
# the stack. For consistency we R's lazy evaluation semantics, we'd
# expect to get the caller of that frame as result:
fn <- function(eval_fn) {
```

```
  list(
    returned_env = middle(eval_fn),
    actual_env = get_env()
  )
}
middle <- function(eval_fn) {
  deep(eval_fn, get_env())
}
deep <- function(eval_fn, eval_env) {
  expr <- quote(parent.frame())
  eval_fn(expr, eval_env)
}

# With eval_bare(), we do get the expected environment:
fn(rlang::eval_bare)

# But that's not the case with base::eval():
fn(base::eval)

# Another difference of eval_bare() compared to base::eval() is
# that it does not insert parasite frames in the evaluation stack:
get_stack <- quote(identity(ctxt_stack()))
eval_bare(get_stack)
eval(get_stack)
```

`eval_tidy` *Evaluate an expression tidily*

Description

`eval_tidy()` is a variant of `base::eval()` and `eval_bare()` that powers the tidy evaluationframework. It evaluates `expr` in an overscope where the special definitions enabling tidy evaluation are installed. This enables the following features:

- Overscoped data. You can supply a data frame or list of named vectors to the `data` argument. The data contained in this list has precedence over the objects in the contextual environment. This is similar to how `base::eval()` accepts a list instead of an environment.
- Self-evaluation of quosures. Within the overscope, quosures act like promises. When a quosure within an expression is evaluated, it automatically invokes the quoted expression in the captured environment (chained to the overscope). Note that quosures do not always get evaluated because of lazy semantics, e.g. `TRUE || ~never_called`.
- Pronouns. `eval_tidy()` installs the `.env` and `.data` pronouns. `.env` contains a reference to the calling environment, while `.data` refers to the `data` argument. These pronouns lets you be explicit about where to find values and throw errors if you try to access non-existent values.

Usage

```
eval_tidy(expr, data = NULL, env = caller_env())
```

Arguments

expr	An expression.
data	A list (or data frame). This is passed to the as_dictionary() coercer, a generic used to transform an object to a proper data source. If you want to make eval_tidy() work for your own objects, you can define a method for this generic.
env	The lexical environment in which to evaluate expr.

See Also

quo(), quasiquotation

Examples

```
# Like base::eval() and eval_bare(), eval_tidy() evaluates quoted
# expressions:
expr <- expr(1 + 2 + 3)
eval_tidy(expr)

# Like base::eval(), it lets you supply overscoping data:
foo <- 1
bar <- 2
expr <- quote(list(foo, bar))
eval_tidy(expr, list(foo = 100))

# The main difference is that quosures self-evaluate within
# eval_tidy():
quo <- quo(1 + 2 + 3)
eval(quo)
eval_tidy(quo)

# Quosures also self-evaluate deep in an expression not just when
# directly supplied to eval_tidy():
expr <- expr(list(list(list(!! quo))))
eval(expr)
eval_tidy(expr)

# Self-evaluation of quosures is powerful because they
# automatically capture their enclosing environment:
foo <- function(x) {
  y <- 10
  quo(x + y)
}
f <- foo(1)

# This quosure refers to `x` and `y` from `foo()`'s evaluation
# frame. That's evaluated consistently by eval_tidy():
f
eval_tidy(f)
```

```
# Finally, eval_tidy() installs handy pronouns that allows users to
# be explicit about where to find symbols. If you supply data,
# eval_tidy() will look there first:
cyl <- 10
eval_tidy(quo(cyl), mtcars)

# To avoid ambiguity and be explicit, you can use the `.env` and
# `.data` pronouns:
eval_tidy(quo(.data$cyl), mtcars)
eval_tidy(quo(.env$cyl), mtcars)

# Note that instead of using `.env` it is often equivalent to
# unquote a value. The only difference is the timing of evaluation
# since unquoting happens earlier (when the quosure is created):
eval_tidy(quo(!! cyl), mtcars)
```

`eval_tidy_` *Tidy evaluation in a custom environment*

Description

We recommend using `eval_tidy()` in your DSLs as much as possible to ensure some consistency across packages (`.data` and `.env` pronouns, etc). However, some DSLs might need a different evaluation environment. In this case, you can call `eval_tidy_()` with the bottom and the top of your custom overscope (see `as_overscope()` for more information).

Usage

```
eval_tidy_(expr, bottom, top = NULL, env = caller_env())
```

Arguments

`expr`	An expression.
`bottom`	This is the environment (or the bottom of a set of environments) containing definitions for overscoped symbols. The bottom environment typically contains pronouns (like `.data`) while its direct parents contain the overscoping bindings. The last one of these parents is the `top`.
`top`	The top environment of the overscope. During tidy evaluation, this environment is chained and rechained to lexical enclosures of self-evaluating formulas (or quosures). This is the mechanism that ensures hygienic scoping: the bindings in the overscope have precedence, but the bindings in the dynamic environment where the tidy quotes were created in the first place are in scope as well.
`env`	The lexical environment in which to evaluate `expr`.

Details

Note that `eval_tidy_()` always installs a `.env` pronoun in the bottom environment of your dynamic scope. This pronoun provides a shortcut to the original lexical enclosure (typically, the dynamic environment of a captured argument, see `enquo()`). It also cleans up the overscope after evaluation. See `overscope_eval_next()` for evaluating several quosures in the same overscope.

`exiting` *Create an exiting or in place handler*

Description

There are two types of condition handlers: exiting handlers, which are thrown to the place where they have been established (e.g., `with_handlers()`'s evaluation frame), and local handlers, which are executed in place (e.g., where the condition has been signalled). `exiting()` and `inplace()` create handlers suitable for `with_handlers()`.

Usage

```
exiting(handler)

inplace(handler, muffle = FALSE)
```

Arguments

`handler`	A handler function that takes a condition as argument. This is passed to `as_function()` and can thus be a formula describing a lambda function.
`muffle`	Whether to muffle the condition after executing an inplace handler. The signalling function must have established a muffling restart. Otherwise, an error will be issued.

Details

A subtle point in the R language is that conditions are not thrown, handlers are. `base::tryCatch()` and `with_handlers()` actually catch handlers rather than conditions. When a critical condition signalled with `base::stop()` or `abort()`, R inspects the handler stack and looks for a handler that can deal with the condition. If it finds an exiting handler, it throws it to the function that established it (`with_handlers()`). That is, it interrupts the normal course of evaluation and jumps to `with_handlers()` evaluation frame (see `ctxt_stack()`), and only then and there the handler is called. On the other hand, if R finds an inplace handler, it executes it locally. The inplace handler can choose to handle the condition by jumping out of the frame (see `rst_jump()` or `return_from()`). If it returns locally, it declines to handle the condition which is passed to the next relevant handler on the stack. If no handler is found or is able to deal with the critical condition (by jumping out of the frame), R will then jump out of the faulty evaluation frame to top-level, via the abort restart (see `rst_abort()`).

See Also

`with_handlers()` for examples, `restarting()` for another kind of inplace handler.

Examples

```
# You can supply a function taking a condition as argument:
hnd <- exiting(function(c) cat("handled foo\n"))
with_handlers(cnd_signal("foo"), foo = hnd)

# Or a lambda-formula where "." is bound to the condition:
with_handlers(foo = inplace(~cat("hello", .$attr, "\n")), {
  cnd_signal("foo", attr = "there")
  "foo"
})
```

---

expr *Raw quotation of an expression*

Description

These functions return raw expressions (whereas `quo()` and variants return quosures). They support quasiquotation syntax.

- `expr()` returns its argument unevaluated. It is equivalent to `base::bquote()`.
- `enexpr()` takes an argument name and returns it unevaluated. It is equivalent to `base::substitute()`.
- `exprs()` captures multiple expressions and returns a list. In particular, it can capture expressions in `...`. It supports name unquoting with `:=` (see `quos()`). It is equivalent to `eval(substitute(alist(...)))`.

See `is_expr()` for more about R expressions.

Usage

```
expr(expr)

enexpr(arg)

exprs(..., .ignore_empty = "trailing")
```

Arguments

`expr`	An expression.
`arg`	A symbol referring to an argument. The expression supplied to that argument will be captured unevaluated.
`...`	Arguments to extract.
`.ignore_empty`	Whether to ignore empty arguments. Can be one of `"trailing"`, `"none"`, `"all"`. If `"trailing"`, only the last argument is ignored if it is empty.

Value

The raw expression supplied as argument. `exprs()` returns a list of expressions.

See Also

`quo()`, `is_expr()`

Examples

```
# The advantage of expr() over quote() is that it unquotes on
# capture:
expr(list(1, !! 3 + 10))

# Unquoting can be especially useful for successive transformation
# of a captured expression:
(expr <- quote(foo(bar)))
(expr <- expr(inner(!! expr, arg1)))
(expr <- expr(outer(!! expr, !!! lapply(letters[1:3], as.symbol))))

# Unlike quo(), expr() produces expressions that can
# be evaluated with base::eval():
e <- quote(letters)
e <- expr(toupper(!!e))
eval(e)

# Be careful if you unquote a quosure: you need to take the RHS
# (and lose the scope information) to evaluate with eval():
f <- quo(letters)
e <- expr(toupper(!! get_expr(f)))
eval(e)

# On the other hand it's fine to unquote quosures if you evaluate
# with eval_tidy():
f <- quo(letters)
e <- expr(toupper(!! f))
eval_tidy(e)

# exprs() lets you unquote names with the definition operator:
nm <- "foo"
exprs(a = 1, !! nm := 2)
```

---

exprs_auto_name *Ensure that list of expressions are all named*

Description

This gives default names to unnamed elements of a list of expressions (or expression wrappers such as formulas or tidy quotes). `exprs_auto_name()` deparses the expressions with `expr_text()` by default. `quos_auto_name()` deparses with `quo_text()`.

Usage

```
exprs_auto_name(exprs, width = 60L, printer = expr_text)

quos_auto_name(quos, width = 60L)
```

Arguments

exprs	A list of expressions.
width	Maximum width of names.
printer	A function that takes an expression and converts it to a string. This function must take an expression as first argument and `width` as second argument.
quos	A list of quosures.

---

expr_interp *Process unquote operators in a captured expression*

Description

While all capturing functions in the tidy evaluation framework perform unquote on capture (most notably `quo()`), `expr_interp()` manually processes unquoting operators in expressions that are already captured. `expr_interp()` should be called in all user-facing functions expecting a formula as argument to provide the same quasiquotation functionality as NSE functions.

Usage

```
expr_interp(x, env = NULL)
```

Arguments

x	A function, raw expression, or formula to interpolate.
env	The environment in which unquoted expressions should be evaluated. By default, the formula or closure environment if a formula or a function, or the current environment otherwise.

Examples

```
# All tidy NSE functions like quo() unquote on capture:
quo(list(!! 1 + 2))

# expr_interp() is meant to provide the same functionality when you
# have a formula or expression that might contain unquoting
# operators:
f <- ~list(!! 1 + 2)
expr_interp(f)

# Note that only the outer formula is unquoted (which is a reason
# to use expr_interp() as early as possible in all user-facing
```

```
# functions):
f <- ~list(~!! 1 + 2, !! 1 + 2)
expr_interp(f)


# Another purpose for expr_interp() is to interpolate a closure's
# body. This is useful to inline a function within another. The
# important limitation is that all formal arguments of the inlined
# function should be defined in the receiving function:
other_fn <- function(x) toupper(x)

fn <- expr_interp(function(x) {
  x <- paste0(x, "_suffix")
  !!! body(other_fn)
})
fn
fn("foo")
```

`expr_label` *Turn an expression to a label*

Description

`expr_text()` turns the expression into a single string, which might be multi-line. `expr_name()` is suitable for formatting names. It works best with symbols and scalar types, but also accepts calls. `expr_label()` formats the expression nicely for use in messages.

Usage

```
expr_label(expr)

expr_name(expr)

expr_text(expr, width = 60L, nlines = Inf)
```

Arguments

`expr`	An expression to labellise.
`width`	Width of each line.
`nlines`	Maximum number of lines to extract.

Examples

```
# To labellise a function argument, first capture it with
# substitute():
fn <- function(x) expr_label(substitute(x))
fn(x:y)

# Strings are encoded
```

```
expr_label("a\nb")

# Names and expressions are quoted with ``
expr_label(quote(x))
expr_label(quote(a + b + c))

# Long expressions are collapsed
expr_label(quote(foo({
  1 + 2
  print(x)
})))
```

`flatten` *Flatten or squash a list of lists into a simpler vector*

Description

`flatten()` removes one level hierarchy from a list, while `squash()` removes all levels. These functions are similar to `unlist()` but they are type-stable so you always know what the type of the output is.

Usage

```
flatten(x)

flatten_lgl(x)

flatten_int(x)

flatten_dbl(x)

flatten_cpl(x)

flatten_chr(x)

flatten_raw(x)

squash(x)

squash_lgl(x)

squash_int(x)

squash_dbl(x)

squash_cpl(x)

squash_chr(x)
```

```
squash_raw(x)

flatten_if(x, predicate = is_spliced)

squash_if(x, predicate = is_spliced)
```

Arguments

x	A list of flatten or squash. The contents of the list can be anything for unsuffixed functions `flatten()` and `squash()` (as a list is returned), but the contents must match the type for the other functions.
predicate	A function of one argument returning whether it should be spliced.

Value

`flatten()` returns a list, `flatten_lgl()` a logical vector, `flatten_int()` an integer vector, `flatten_dbl()` a double vector, and `flatten_chr()` a character vector. Similarly for `squash()` and the typed variants (`squash_lgl()` etc).

Examples

```
x <- replicate(2, sample(4), simplify = FALSE)
x

flatten(x)
flatten_int(x)

# With flatten(), only one level gets removed at a time:
deep <- list(1, list(2, list(3)))
flatten(deep)
flatten(flatten(deep))

# But squash() removes all levels:
squash(deep)
squash_dbl(deep)

# The typed flattens remove one level and coerce to an atomic
# vector at the same time:
flatten_dbl(list(1, list(2)))

# Only bare lists are flattened, but you can splice S3 lists
# explicitly:
foo <- set_attrs(list("bar"), class = "foo")
str(flatten(list(1, foo, list(100))))
str(flatten(list(1, splice(foo), list(100))))

# Instead of splicing manually, flatten_if() and squash_if() let
# you specify a predicate function:
is_foo <- function(x) inherits(x, "foo") || is_bare_list(x)
str(flatten_if(list(1, foo, list(100)), is_foo))
```

```
# squash_if() does the same with deep lists:
deep_foo <- list(1, list(foo, list(foo, 100)))
str(deep_foo)

str(squash(deep_foo))
str(squash_if(deep_foo, is_foo))
```

`fn_env` *Return the closure environment of a function*

Description

Closure environments define the scope of functions (see `env()`). When a function call is evaluated, R creates an evaluation frame (see `ctxt_stack()`) that inherits from the closure environment. This makes all objects defined in the closure environment and all its parents available to code executed within the function.

Usage

```
fn_env(fn)

fn_env(x) <- value
```

Arguments

`fn, x`	A function.
`value`	A new closure environment for the function.

Details

`fn_env()` returns the closure environment of `fn`. There is also an assignment method to set a new closure environment.

Examples

```
env <- child_env("base")
fn <- with_env(env, function() NULL)
identical(fn_env(fn), env)

other_env <- child_env("base")
fn_env(fn) <- other_env
identical(fn_env(fn), other_env)
```

`fn_fmls` *Extract arguments from a function*

Description

`fn_fmls()` returns a named list of formal arguments. `fn_fmls_names()` returns the names of the arguments. `fn_fmls_syms()` returns formals as a named list of symbols. This is especially useful for forwarding arguments in constructed calls.

Usage

```
fn_fmls(fn = caller_fn())

fn_fmls_names(fn = caller_fn())

fn_fmls_syms(fn = caller_fn())
```

Arguments

`fn`	A function. It is lookep up in the calling frame if not supplied.

Details

Unlike `formals()`, these helpers also work with primitive functions. See `is_function()` for a discussion of primitive and closure functions.

See Also

`lang_args()` and `lang_args_names()`

Examples

```
# Extract from current call:
fn <- function(a = 1, b = 2) fn_fmls()
fn()

# Works with primitive functions:
fn_fmls(base::switch)

# fn_fmls_syms() makes it easy to forward arguments:
lang("apply", !!! fn_fmls_syms(lapply))
```

`frame_position` *Find the position or distance of a frame on the evaluation stack*

Description

The frame position on the stack can be computed by counting frames from the global frame (the bottom of the stack, the default) or from the current frame (the top of the stack).

Usage

```
frame_position(frame, from = c("global", "current"))
```

Arguments

`frame`	The environment of a frame. Can be any object with a `get_env()` method. Note that for frame objects, the position from the global frame is simply `frame$pos`. Alternatively, `frame` can be an integer that represents the position on the stack (and is thus returned as is if `from` is "global".
`from`	Whether to compute distance from the global frame (the bottom of the evaluation stack), or from the current frame (the top of the evaluation stack).

Details

While this function returns the position of the frame on the evaluation stack, it can safely be called with intervening frames as those will be discarded.

Examples

```
fn <- function() g(environment())
g <- function(env) frame_position(env)

# frame_position() returns the position of the frame on the evaluation stack:
fn()
identity(identity(fn()))

# Note that it trims off intervening calls before counting so you
# can safely nest it within other calls:
g <- function(env) identity(identity(frame_position(env)))
fn()

# You can also ask for the position from the current frame rather
# than the global frame:
fn <- function() g(environment())
g <- function(env) h(env)
h <- function(env) frame_position(env, from = "current")
fn()
```

friendly_type *Format a type for error messages*

Description

Format a type for error messages

Usage

```
friendly_type(type)
```

Arguments

type	A type as returned by `type_of()` or `lang_type_of()`.

Value

A string of the prettified type, qualified with an indefinite article.

Examples

```
friendly_type("logical")
friendly_type("integer")
friendly_type("string")
```

f_rhs *Get or 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 *Turn RHS of formula into a string or label*

Description

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

Usage

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

f_name(x)

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)
}))
```

---

get_env *Get or set the environment of an object*

Description

These functions dispatch internally with methods for functions, formulas and frames. If called with a missing argument, the environment of the current evaluation frame (see `ctxt_stack()`) is returned. If you call `get_env()` with an environment, it acts as the identity function and the environment is simply returned (this helps simplifying code when writing generic functions for environments).

Usage

```
get_env(env = caller_env(), default = NULL)

set_env(env, new_env = caller_env())
```

Arguments

`env`	An environment or an object bundling an environment, e.g. a formula, quosure or closure.
`default`	The default environment in case `env` does not wrap an environment. If `NULL` and no environment could be extracted, an error is issued.
`new_env`	An environment to replace `env` with. Can be an object handled by `get_env()`.

Examples

```
# Get the environment of frame objects. If no argument is supplied,
# the current frame is used:
fn <- function() {
  list(
    get_env(call_frame()),
    get_env()
```

```
  )
}
fn()

# Environment of closure functions:
get_env(fn)

# Or of quosures or formulas:
get_env(~foo)
get_env(quo(foo))


# Provide a default in case the object doesn't bundle an environment.
# Let's create an unevaluated formula:
f <- quote(~foo)

# The following line would fail if run because unevaluated formulas
# don't bundle an environment (they didn't have the chance to
# record one yet):
# get_env(f)

# It is often useful to provide a default when you're writing
# functions accepting formulas as input:
default <- env()
identical(get_env(f, default), default)

# set_env() can be used to set the enclosure of functions and
# formulas. Let's create a function with a particular environment:
env <- child_env("base")
fn <- set_env(function() NULL, env)

# That function now has `env` as enclosure:
identical(get_env(fn), env)
identical(get_env(fn), get_env())

# set_env() does not work by side effect. Setting a new environment
# for fn has no effect on the original function:
other_env <- child_env(NULL)
set_env(fn, other_env)
identical(get_env(fn), other_env)

# Since set_env() returns a new function with a different
# environment, you'll need to reassign the result:
fn <- set_env(fn, other_env)
identical(get_env(fn), other_env)
```

---

has_length *How long is an object?*

Description

This is a function for the common task of testing the length of an object. It checks the length of an object in a non-generic way: `base::length()` methods are ignored.

Usage

```
has_length(x, n = NULL)
```

Arguments

x	A R object.
n	A specific length to test x with. If NULL, has_length() returns TRUE if x has length greater than zero, and FALSE otherwise.

Examples

```
has_length(list())
has_length(list(), 0)

has_length(letters)
has_length(letters, 20)
has_length(letters, 26)
```

---

has_name *Does an object have an element with this name?*

Description

This function returns a logical value that indicates if a data frame or another named object contains an element with a specific name.

Usage

```
has_name(x, name)
```

Arguments

x	A data frame or another named object
name	Element name(s) to check

Details

Unnamed objects are treated as if all names are empty strings. NA input gives FALSE as output.

Value

A logical vector of the same length as name

Examples

```
has_name(iris, "Species")
has_name(mtcars, "gears")
```

---

invoke *Invoke a function with a list of arguments*

Description

Normally, you invoke a R function by typing arguments manually. A powerful alternative is to call a function with a list of arguments assembled programmatically. This is the purpose of `invoke()`.

Usage

```
invoke(.fn, .args = list(), ..., .env = caller_env(), .bury = c(".fn",
  ""))
```

Arguments

`.fn`	A function to invoke. Can be a function object or the name of a function in scope of `.env`.
`.args, ...`	List of arguments (possibly named) to be passed to `.fn`.
`.env`	The environment in which to call `.fn`.
`.bury`	A character vector of length 2. The first string specifies which name should the function have in the call recorded in the evaluation stack. The second string specifies a prefix for the argument names. Set `.bury` to `NULL` if you prefer to inline the function and its arguments in the call.

Details

Technically, `invoke()` is basically a version of `base::do.call()` that creates cleaner call traces because it does not inline the function and the arguments in the call (see examples). To achieve this, `invoke()` creates a child environment of `.env` with `.fn` and all arguments bound to new symbols (see `env_bury()`). It then uses the same strategy as `eval_bare()` to evaluate with minimal noise.

Examples

```
# invoke() has the same purpose as do.call():
invoke(paste, letters)

# But it creates much cleaner calls:
invoke(call_inspect, mtcars)

# and stacktraces:
fn <- function(...) sys.calls()
invoke(fn, list(mtcars))
```

```
# Compare to do.call():
do.call(call_inspect, mtcars)
do.call(fn, list(mtcars))


# Specify the function name either by supplying a string
# identifying the function (it should be visible in .env):
invoke("call_inspect", letters)

# Or by changing the .bury argument, with which you can also change
# the argument prefix:
invoke(call_inspect, mtcars, .bury = c("inspect!", "col"))
```

`is_callable` *Is an object callable?*

Description

A callable object is an object that can be set as the head of a call node. This includes symbolic objects that evaluate to a function or literal functions.

Usage

```
is_callable(x)
```

Arguments

x	An object to test.

Details

Note that strings may look like callable objects because expressions of the form `"list"()` are valid R code. However, that's only because the R parser transforms strings to symbols. It is not legal to manually set language heads to strings.

Examples

```
# Symbolic objects and functions are callable:
is_callable(quote(foo))
is_callable(base::identity)

# mut_node_car() lets you modify calls without any checking:
lang <- quote(foo(10))
mut_node_car(lang, get_env())

# Use is_callable() to check an input object is safe to put as CAR:
obj <- base::identity

if (is_callable(obj)) {
  lang <- mut_node_car(lang, obj)
```

```
} else {
  abort("`obj` must be callable")
}

eval_bare(lang)
```

`is_condition` *Is object a condition?*

Description

Is object a condition?

Usage

```
is_condition(x)
```

Arguments

`x`	An object to test.

`is_copyable` *Is an object copyable?*

Description

When an object is modified, R generally copies it (sometimes lazily) to enforce valuesemantics. However, some internal types are uncopyable. If you try to copy them, either with `<-` or by argument passing, you actually create references to the original object rather than actual copies. Modifying these references can thus have far reaching side effects.

Usage

```
is_copyable(x)
```

Arguments

`x`	An object to test.

Examples

```
# Let's add attributes with structure() to uncopyable types. Since
# they are not copied, the attributes are changed in place:
env <- env()
structure(env, foo = "bar")
env

# These objects that can only be changed with side effect are not
# copyable:
is_copyable(env)

structure(base::list, foo = "bar")
str(base::list)
```

---

is_empty *Is object an empty vector or NULL?*

Description

Is object an empty vector or NULL?

Usage

```
is_empty(x)
```

Arguments

x	object to test

Examples

```
is_empty(NULL)
is_empty(list())
is_empty(list(NULL))
```

---

is_env *Is object an environment?*

Description

`is_bare_env()` tests whether `x` is an environment without a s3 or s4 class.

Usage

```
is_env(x)

is_bare_env(x)
```

Arguments

x	object to test

`is_expr` *Is an object an expression?*

Description

`is_expr()` tests for expressions, the set of objects that can be obtained from parsing R code. An expression can be one of two things: either a symbolic object (for which `is_symbolic()` returns `TRUE`), or a syntactic literal (testable with `is_syntactic_literal()`). Technically, calls can contain any R object, not necessarily symbolic objects or syntactic literals. However, this only happens in artificial situations. Expressions as we define them only contain numbers, strings, `NULL`, symbols, and calls: this is the complete set of R objects that can be created when R parses source code (e.g. from using `parse_expr()`).

Note that we are using the term expression in its colloquial sense and not to refer to `expression()` vectors, a data type that wraps expressions in a vector and which isn't used much in modern R code.

Usage

```
is_expr(x)

is_syntactic_literal(x)

is_symbolic(x)
```

Arguments

x	An object to test.

Details

`is_symbolic()` returns `TRUE` for symbols and calls (objects with type `language`). Symbolic objects are replaced by their value during evaluation. Literals are the complement of symbolic objects. They are their own value and return themselves during evaluation.

`is_syntactic_literal()` is a predicate that returns `TRUE` for the subset of literals that are created by R when parsing text (see `parse_expr()`): numbers, strings and `NULL`. Along with symbols, these literals are the terminating nodes in a parse tree.

Note that in the most general sense, a literal is any R object that evaluates to itself and that can be evaluated in the empty environment. For instance, `quote(c(1, 2))` is not a literal, it is a call. However, the result of evaluating it in `base_env()` is a literal(in this case an atomic vector).

Pairlists are also a kind of language objects. However, since they are mostly an internal data structure, `is_expr()` returns `FALSE` for pairlists. You can use `is_pairlist()` to explicitly check for them. Pairlists are the data structure for function arguments. They usually do not arise from R code because subsetting a call is a type-preserving operation. However, you can obtain the pairlist of

arguments by taking the CDR of the call object from C code. The rlang function `lang_tail()` will do it from R. Another way in which pairlist of arguments arise is by extracting the argument list of a closure with `base::formals()` or `fn_fmls()`.

See Also

`is_lang()` for a call predicate.

Examples

```
q1 <- quote(1)
is_expr(q1)
is_syntactic_literal(q1)

q2 <- quote(x)
is_expr(q2)
is_symbol(q2)

q3 <- quote(x + 1)
is_expr(q3)
is_lang(q3)


# Atomic expressions are the terminating nodes of a call tree:
# NULL or a scalar atomic vector:
is_syntactic_literal("string")
is_syntactic_literal(NULL)

is_syntactic_literal(letters)
is_syntactic_literal(quote(call()))

# Parsable literals have the property of being self-quoting:
identical("foo", quote("foo"))
identical(1L, quote(1L))
identical(NULL, quote(NULL))

# Like any literals, they can be evaluated within the empty
# environment:
eval_bare(quote(1L), empty_env())

# Whereas it would fail for symbolic expressions:
# eval_bare(quote(c(1L, 2L)), empty_env())


# Pairlists are also language objects representing argument lists.
# You will usually encounter them with extracted formals:
fmls <- formals(is_expr)
typeof(fmls)

# Since they are mostly an internal data structure, is_expr()
# returns FALSE for pairlists, so you will have to check explicitly
# for them:
```

```
is_expr(fmls)
is_pairlist(fmls)

# Note that you can also extract call arguments as a pairlist:
lang_tail(quote(fn(arg1, arg2 = "foo")))
```

`is_formula` *Is object a formula?*

Description

`is_formula()` tests if `x` is a call to `~`. `is_bare_formula()` tests in addition that `x` does not inherit from anything else than `"formula"`. `is_formulaish()` returns `TRUE` for both formulas and definitions of the type `a := b`.

Usage

```
is_formula(x, scoped = NULL, lhs = NULL)

is_bare_formula(x, scoped = NULL, lhs = NULL)

is_formulaish(x, scoped = NULL, lhs = NULL)
```

Arguments

`x`	An object to test.
`scoped`	A boolean indicating whether the quosure or formula is scoped, that is, has a valid environment attribute. If `NULL`, the scope is not inspected.
`lhs`	A boolean indicating whether the formula or definition has a left-hand side. If `NULL`, the LHS is not inspected.

Details

The `scoped` argument patterns-match on whether the scoped bundled with the quosure is valid or not. Invalid scopes may happen in nested quotations like `~~expr`, where the outer quosure is validly scoped but not the inner one. This is because `~` saves the environment when it is evaluated, and quoted formulas are by definition not evaluated.

See Also

`is_quosure()` and `is_quosureish()`

Examples

```
x <- disp ~ am
is_formula(x)

is_formula(~10)
is_formula(10)

is_formula(quo(foo))
is_bare_formula(quo(foo))

# Note that unevaluated formulas are treated as bare formulas even
# though they don't inherit from "formula":
f <- quote(~foo)
is_bare_formula(f)

# However you can specify `scoped` if you need the predicate to
# return FALSE for these unevaluated formulas:
is_bare_formula(f, scoped = TRUE)
is_bare_formula(eval(f), scoped = TRUE)


# There is also a variant that returns TRUE for definitions in
# addition to formulas:
is_formulaish(a ~ b)
is_formulaish(a := b)
```

---

is_frame *Is object a frame?*

Description

Is object a frame?

Usage

```
is_frame(x)
```

Arguments

x	Object to test

`is_function` *Is object a function?*

Description

The R language defines two different types of functions: primitive functions, which are low-level, and closures, which are the regular kind of functions.

Usage

```
is_function(x)

is_closure(x)

is_primitive(x)

is_primitive_eager(x)

is_primitive_lazy(x)
```

Arguments

`x`	Object to be tested.

Details

Closures are functions written in R, named after the way their arguments are scoped within nested environments (see https://en.wikipedia.org/wiki/Closure_(computer_programming)). The root environment of the closure is called the closure environment. When closures are evaluated, a new environment called the evaluation frame is created with the closure environment as parent. This is where the body of the closure is evaluated. These closure frames appear on the evaluation stack (see `ctxt_stack()`), as opposed to primitive functions which do not necessarily have their own evaluation frame and never appear on the stack.

Primitive functions are more efficient than closures for two reasons. First, they are written entirely in fast low-level code. Secondly, the mechanism by which they are passed arguments is more efficient because they often do not need the full procedure of argument matching (dealing with positional versus named arguments, partial matching, etc). One practical consequence of the special way in which primitives are passed arguments this is that they technically do not have formal arguments, and `formals()` will return `NULL` if called on a primitive function. See `fn_fmls()` for a function that returns a representation of formal arguments for primitive functions. Finally, primitive functions can either take arguments lazily, like R closures do, or evaluate them eagerly before being passed on to the C code. The former kind of primitives are called "special" in R terminology, while the latter is referred to as "builtin". `is_primitive_eager()` and `is_primitive_lazy()` allow you to check whether a primitive function evaluates arguments eagerly or lazily.

You will also encounter the distinction between primitive and internal functions in technical documentation. Like primitive functions, internal functions are defined at a low level and written in C.

However, internal functions have no representation in the R language. Instead, they are called via a call to `base::.Internal()` within a regular closure. This ensures that they appear as normal R function objects: they obey all the usual rules of argument passing, and they appear on the evaluation stack as any other closures. As a result, `fn_fmls()` does not need to look in the `.ArgsEnv` environment to obtain a representation of their arguments, and there is no way of querying from R whether they are lazy ('special' in R terminology) or eager ('builtin').

You can call primitive functions with `.Primitive()` and internal functions with `.Internal()`. However, calling internal functions in a package is forbidden by CRAN's policy because they are considered part of the private API. They often assume that they have been called with correctly formed arguments, and may cause R to crash if you call them with unexpected objects.

Examples

```
# Primitive functions are not closures:
is_closure(base::c)
is_primitive(base::c)

# On the other hand, internal functions are wrapped in a closure
# and appear as such from the R side:
is_closure(base::eval)

# Both closures and primitives are functions:
is_function(base::c)
is_function(base::eval)

# Primitive functions never appear in evaluation stacks:
is_primitive(base::`[[`)
is_primitive(base::list)
list(ctxt_stack())[[1]]

# While closures do:
identity(identity(ctxt_stack()))

# Many primitive functions evaluate arguments eagerly:
is_primitive_eager(base::c)
is_primitive_eager(base::list)
is_primitive_eager(base::`+`)

# However, primitives that operate on expressions, like quote() or
# substitute(), are lazy:
is_primitive_lazy(base::quote)
is_primitive_lazy(base::substitute)
```

`is_installed` *Is a package installed in the library?*

Description

This checks that a package is installed with minimal side effects. If installed, the package will be loaded but not attached.

Usage

```
is_installed(pkg)
```

Arguments

pkg	The name of a package.

Value

TRUE if the package is installed, FALSE otherwise.

Examples

```
is_installed("utils")
is_installed("ggplot5")
```

---

is_integerish *Is a vector integer-like?*

Description

These predicates check whether R considers a number vector to be integer-like, according to its own tolerance check (which is in fact delegated to the C library). This function is not adapted to data analysis, see the help for base::is.integer() for examples of how to check for whole numbers.

Usage

```
is_integerish(x, n = NULL)

is_bare_integerish(x, n = NULL)

is_scalar_integerish(x)
```

Arguments

x	Object to be tested.
n	Expected length of a vector.

See Also

is_bare_numeric() for testing whether an object is a base numeric type (a bare double or integer vector).

Examples

```
is_integerish(10L)
is_integerish(10.0)
is_integerish(10.000001)
is_integerish(TRUE)
```

is_lang *Is object a call?*

Description

This function tests if x is a call (or language object). This is a pattern-matching predicate that will return FALSE if name and n are supplied and the call does not match these properties. is_unary_lang() and is_binary_lang() hardcode n to 1 and 2.

Usage

```
is_lang(x, name = NULL, n = NULL, ns = NULL)

is_unary_lang(x, name = NULL, ns = NULL)

is_binary_lang(x, name = NULL, ns = NULL)
```

Arguments

x	An object to test. If a formula, the right-hand side is extracted.
name	An optional name that the call should match. It is passed to sym() before matching. This argument is vectorised and you can supply a vector of names to match. In this case, is_lang() returns TRUE if at least one name matches.
n	An optional number of arguments that the call should match.
ns	The namespace of the call. If NULL, the namespace doesn't participate in the pattern-matching. If an empty string "" and x is a namespaced call, is_lang() returns FALSE. If any other string, is_lang() checks that x is namespaced within ns.

See Also

is_expr()

Examples

```
is_lang(quote(foo(bar)))

# You can pattern-match the call with additional arguments:
is_lang(quote(foo(bar)), "foo")
is_lang(quote(foo(bar)), "bar")
is_lang(quote(foo(bar)), quote(foo))

# Match the number of arguments with is_lang():
is_lang(quote(foo(bar)), "foo", 1)
is_lang(quote(foo(bar)), "foo", 2)

# Or more specifically:
```

```
is_unary_lang(quote(foo(bar)))
is_unary_lang(quote(+3))
is_unary_lang(quote(1 + 3))
is_binary_lang(quote(1 + 3))


# By default, namespaced calls are tested unqualified:
ns_expr <- quote(base::list())
is_lang(ns_expr, "list")

# You can also specify whether the call shouldn't be namespaced by
# supplying an empty string:
is_lang(ns_expr, "list", ns = "")

# Or if it should have a namespace:
is_lang(ns_expr, "list", ns = "utils")
is_lang(ns_expr, "list", ns = "base")


# The name argument is vectorised so you can supply a list of names
# to match with:
is_lang(quote(foo(bar)), c("bar", "baz"))
is_lang(quote(foo(bar)), c("bar", "foo"))
is_lang(quote(base::list), c("::", ":::", "$", "@"))
```

`is_named` *Is object named?*

Description

`is_named()` checks that `x` has names attributes, and that none of the names are missing or empty (`NA` or `""`). `is_dictionaryish()` checks that an object is a dictionary: that it has actual names and in addition that there are no duplicated names. `have_name()` is a vectorised version of `is_named()`.

Usage

```
is_named(x)

is_dictionaryish(x)

have_name(x)
```

Arguments

`x`	An object to test.

Value

`is_named()` and `is_dictionaryish()` are scalar predicates and return `TRUE` or `FALSE`. `have_name()` is vectorised and returns a logical vector as long as the input.

Examples

```
# A data frame usually has valid, unique names
is_named(mtcars)
have_name(mtcars)
is_dictionaryish(mtcars)

# But data frames can also have duplicated columns:
dups <- cbind(mtcars, cyl = seq_len(nrow(mtcars)))
is_dictionaryish(dups)

# The names are still valid:
is_named(dups)
have_name(dups)


# For empty objects the semantics are slightly different.
# is_dictionaryish() returns TRUE for empty objects:
is_dictionaryish(list())

# But is_named() will only return TRUE if there is a names
# attribute (a zero-length character vector in this case):
x <- set_names(list(), character(0))
is_named(x)


# Empty and missing names are invalid:
invalid <- dups
names(invalid)[2] <- ""
names(invalid)[5] <- NA

# is_named() performs a global check while have_name() can show you
# where the problem is:
is_named(invalid)
have_name(invalid)

# have_name() will work even with vectors that don't have a names
# attribute:
have_name(letters)
```

`is_pairlist` *Is object a node or pairlist?*

Description

- `is_pairlist()` checks that `x` has type `pairlist` or `NULL`. `NULL` is treated as a pairlist because it is the terminating node of pairlists and an empty pairlist is thus the `NULL` object itself.
- `is_node()` checks that `x` has type `pairlist`.

In other words, `is_pairlist()` tests for the data structure while `is_node()` tests for the internal type.

Usage

```
is_pairlist(x)

is_node(x)
```

Arguments

x	Object to test.

See Also

is_lang() tests for language nodes.

---

is_quosure *Is an object a quosure or quosure-like?*

Description

These predicates test for quosure objects.

- is_quosure() tests for a tidyeval quosure. These are one-sided formulas with a quosure class.
- is_quosureish() tests for general R quosure objects: quosures, or one-sided formulas.

Usage

```
is_quosure(x)

is_quosureish(x, scoped = NULL)
```

Arguments

x	An object to test.
scoped	A boolean indicating whether the quosure or formula is scoped, that is, has a valid environment attribute. If NULL, the scope is not inspected.

See Also

is_formula() and is_formulaish()

Examples

```
# Quosures are created with quo():
quo(foo)
is_quosure(quo(foo))

# Formulas look similar to quosures but are not quosures:
is_quosure(~foo)

# But they are quosureish:
is_quosureish(~foo)

# Note that two-sided formulas are never quosureish:
is_quosureish(a ~ b)
```

---

is_stack *Is object a stack?*

Description

Is object a stack?

Usage

```
is_stack(x)

is_eval_stack(x)

is_call_stack(x)
```

Arguments

x	An object to test

---

is_symbol *Is object a symbol?*

Description

Is object a symbol?

Usage

```
is_symbol(x)
```

Arguments

x	An object to test.

is_true *Is object identical to TRUE or FALSE?*

Description

These functions bypass R's automatic conversion rules and check that `x` is literally `TRUE` or `FALSE`.

Usage

```
is_true(x)

is_false(x)
```

Arguments

`x`	object to test

Examples

```
is_true(TRUE)
is_true(1)

is_false(FALSE)
is_false(0)
```

lang *Create a call*

Description

Language objects are (with symbols) one of the two types of symbolic objects in R. These symbolic objects form the backbone of expressions. They represent a value, unlike literal objects which are their own values. While symbols are directly bound to a value, language objects represent *function calls*, which is why they are commonly referred to as calls.

- `lang()` creates a call from a function name (or a literal function to inline in the call) and a list of arguments.
- `new_language()` is bare-bones and takes a head and a tail. The head must be callable and the tail must be a pairlist. See section on calls as parse trees below. This constructor is useful to avoid costly coercions between lists and pairlists of arguments.

Usage

```
lang(.fn, ..., .ns = NULL)

new_language(head, tail = NULL)
```

Arguments

`.fn`	Function to call. Must be a callable object: a string, symbol, call, or a function.
`...`	Arguments to the call either in or out of a list. Dots are evaluated with explicit splicing.
`.ns`	Namespace with which to prefix `.fn`. Must be a string or symbol.
`head`	A callable object: a symbol, call, or literal function.
`tail`	A pairlist of arguments.

Calls as parse tree

Language objects are structurally identical to pairlists. They are containers of two objects, the head and the tail (also called the CAR and the CDR).

- The head contains the function to call, either literally or symbolically. If a literal function, the call is said to be inlined. If a symbol, the call is named. If another call, it is recursive. `foo()()` would be an example of a recursive call whose head contains another call. See `lang_type_of()` and `is_callable()`.
- The tail contains the arguments and must be a pairlist.

You can retrieve those components with `lang_head()` and `lang_tail()`. Since language nodes can contain other nodes (either calls or pairlists), they are capable of forming a tree. When R parses an expression, it saves the parse tree in a data structure composed of language and pairlist nodes. It is precisely because the parse tree is saved in first-class R objects that it is possible for functions to capture their arguments unevaluated.

Call versus language

`call` is the old S mode of these objects while `language` is the R type. While it is usually better to avoid using S terminology, it would probably be even more confusing to systematically refer to "calls" as "language". rlang still uses `lang` as particle for function dealing with calls for consistency.

See Also

lang_modify

Examples

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

#' Can supply arguments individually or in a list
lang(quote(f), a = 1, b = 2)
lang(quote(f), splice(list(a = 1, b = 2)))

# Creating namespaced calls:
lang("fun", arg = quote(baz), .ns = "mypkg")
```

lang_args *Extract arguments from a call*

Description

Extract arguments from a call

Usage

```
lang_args(lang)

lang_args_names(lang)
```

Arguments

lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.

Value

A named list of arguments.

See Also

lang_tail(), fn_fmls() and fn_fmls_names()

Examples

```
call <- quote(f(a, b))

# Subsetting a call returns the arguments converted to a language
# object:
call[-1]

# See also lang_tail() which returns the arguments without
# conversion as the original pairlist:
str(lang_tail(call))

# On the other hand, lang_args() returns a regular list that is
# often easier to work with:
str(lang_args(call))

# When the arguments are unnamed, a vector of empty strings is
# supplied (rather than NULL):
lang_args_names(call)
```

lang_fn *Extract function from a call*

Description

If a frame or formula, the function will be retrieved from the associated environment. Otherwise, it is looked up in the calling frame.

Usage

```
lang_fn(lang)
```

Arguments

lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.

See Also

lang_name()

Examples

```
# Extract from a quoted call:
lang_fn(~matrix())
lang_fn(quote(matrix()))

# Extract the calling function
test <- function() lang_fn(call_frame())
test()
```

lang_head *Return the head or tail of a call object*

Description

These functions return the head or the tail of a call. See section on calls as parse trees in lang(). They are equivalent to node_car() and node_cdr() but support quosures and check that the input is indeed a call before retrieving the head or tail (it is unsafe to do this without type checking).

lang_head() returns the head of the call without any conversion, unlike lang_name() which checks that the head is a symbol and converts it to a string. lang_tail() returns the pairlist of arguments (while lang_args() returns the same object converted to a regular list)

Usage

```
lang_head(lang)

lang_tail(lang)
```

Arguments

lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.

See Also

pairlist, `lang_args()`, `lang()`

Examples

```
lang <- quote(foo(bar, baz))
lang_head(lang)
lang_tail(lang)
```

---

lang_modify *Modify the arguments of a call*

Description

Modify the arguments of a call

Usage

```
lang_modify(.lang, ..., .standardise = FALSE)
```

Arguments

.lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.
...	Named or unnamed expressions (constants, names or calls) used to modify the call. Use NULL to remove arguments. Dots are evaluated with explicit splicing.
.standardise	If TRUE, the call is standardised before hand to match existing unnamed arguments to their argument names. This prevents new named arguments from accidentally replacing original unnamed arguments.

Value

A quosure if .lang is a quosure, a call otherwise.

See Also

lang

Examples

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

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

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

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

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

# Supply a list of new arguments with splice()
newargs <- list(na.rm = NULL, trim = 0.1)
lang_modify(call, splice(newargs))

# Supply a call frame to extract the frame expression:
f <- function(bool = TRUE) {
  lang_modify(call_frame(), splice(list(bool = FALSE)))
}
f()


# You can also modify quosures inplace:
f <- ~matrix(bar)
lang_modify(f, quote(foo))
```

---

lang_name *Extract function name of a call*

Description

Extract function name of a call

Usage

```
lang_name(lang)
```

Arguments

lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.

Value

A string with the function name, or NULL if the function is anonymous.

See Also

lang_fn()

Examples

```
# Extract the function name from quoted calls:
lang_name(~foo(bar))
lang_name(quote(foo(bar)))

# Or from a frame:
foo <- function(bar) lang_name(call_frame())
foo(bar)

# Namespaced calls are correctly handled:
lang_name(~base::matrix(baz))

# Anonymous and subsetted functions return NULL:
lang_name(~foo$bar())
lang_name(~foo[[bar]]())
lang_name(~foo()())
```

---

lang_standardise *Standardise a call*

Description

This is essentially equivalent to base::match.call(), but with better handling of primitive functions.

Usage

```
lang_standardise(lang)
```

Arguments

lang	Can be a call (language object), a formula quoting a call in the right-hand side, or a frame object from which to extract the call expression.

Value

A quosure if .lang is a quosure, a raw call otherwise.

`missing` *Missing values*

Description

Missing values are represented in R with the general symbol `NA`. They can be inserted in almost all data containers: all atomic vectors except raw vectors can contain missing values. To achieve this, R automatically converts the general `NA` symbol to a typed missing value appropriate for the target vector. The objects provided here are aliases for those typed `NA` objects.

Usage

```
na_lgl

na_int

na_dbl

na_chr

na_cpl
```

Format

An object of class `logical` of length 1.

Details

Typed missing values are necessary because R needs sentinel values of the same type (i.e. the same machine representation of the data) as the containers into which they are inserted. The official typed missing values are `NA_integer_`, `NA_real_`, `NA_character_` and `NA_complex_`. The missing value for logical vectors is simply the default `NA`. The aliases provided in rlang are consistently named and thus simpler to remember. Also, `na_lgl` is provided as an alias to `NA` that makes intent clearer.

Since `na_lgl` is the default `NA`, expressions such as `c(NA, NA)` yield logical vectors as no data is available to give a clue of the target type. In the same way, since lists and environments can contain any types, expressions like `list(NA)` store a logical `NA`.

See Also

The vector-along family to create typed vectors filled with missing values.

Examples

```
typeof(NA)
typeof(na_lgl)
typeof(na_int)

# Note that while the base R missing symbols cannot be overwritten,
```

```
# that's not the case for rlang's aliases:
na_dbl <- NA
typeof(na_dbl)
```

`missing_arg` *Generate or handle a missing argument*

Description

These functions help using the missing argument as a regular R object. It is valid to generate a missing argument and assign it in the current environment or in a list. However, once assigned in the environment, the missing argument normally cannot be touched. `maybe_missing()` checks whether the object is the missing argument, and regenerate it if needed to prevent R from throwing a missing error. In addition, `is_missing()` lets you check for a missing argument in a larger range of situations than `base::missing()` (see examples).

Usage

```
missing_arg()

is_missing(x)

maybe_missing(x)
```

Arguments

`x`	An object that might be the missing argument.

Examples

```
# The missing argument can be useful to generate calls
quo(f(x = !! missing_arg()))
quo(f(x = !! NULL))


# It is perfectly valid to generate and assign the missing
# argument.
x <- missing_arg()
l <- list(missing_arg())

# Note that accessing a missing argument contained in a list does
# not trigger an error:
l[[1]]
is.null(l[[1]])

# But if the missing argument is assigned in the current
# environment, it is no longer possible to touch it. The following
# lines would all return errors:
#> x
```

```
#> is.null(x)

# In these cases, you can use maybe_missing() to manipulate an
# object that might be the missing argument without triggering a
# missing error:
maybe_missing(x)
is.null(maybe_missing(x))
is_missing(maybe_missing(x))


# base::missing() does not work well if you supply an
# expression. The following lines would throw an error:

#> missing(missing_arg())
#> missing(l[[1]])

# while is_missing() will work as expected:
is_missing(missing_arg())
is_missing(l[[1]])
```

`modify` *Modify a vector*

Description

This function merges a list of arguments into a vector. It always returns a list.

Usage

```
modify(.x, ...)
```

Arguments

`.x`	A vector to modify.
`...`	List of elements to merge into `.x`. Named elements already existing in `.x` are used as replacements. Elements that have new or no names are inserted at the end. These dots are evaluated with explicit splicing.

Value

A modified vector upcasted to a list.

Examples

```
modify(c(1, b = 2, 3), 4, b = "foo")

x <- list(a = 1, b = 2)
y <- list(b = 3, c = 4)
modify(x, splice(y))
```

mut_utf8_locale *Set the locale's codeset for testing*

Description

Setting a locale's codeset (specifically, the LC_CTYPE category) produces side effects in R's handling of strings. The most important of these affects how the R parser marks strings. R has specific internal support for latin1 (single-byte encoding) and UTF-8 (multi-bytes variable-width encoding) strings. If the locale codeset is latin1 or UTF-8, the parser will mark all strings with the corresponding encoding. It is important for strings to have consistent encoding markers, as they determine a number of internal encoding conversions when R or packages handle strings (see set_str_encoding() for some examples).

Usage

```
mut_utf8_locale()

mut_latin1_locale()

mut_mbcs_locale()
```

Details

If you are changing the locale encoding for testing purposes, you need to be aware that R caches strings and symbols to save memory. If you change the locale during an R session, it can lead to surprising and difficult to reproduce results. In doubt, restart your R session.

Note that these helpers are only provided for testing interactively the effects of changing locale codeset. They let you quickly change the default text encoding to latin1, UTF-8, or non-UTF-8 MBCS. They are not widely tested and do not provide a way of setting the language and region of the locale. They have permanent side effects and should probably not be used in package examples, unit tests, or in the course of a data analysis. Note finally that mut_utf8_locale() will not work on Windows as only latin1 and MBCS locales are supported on this OS.

Value

The previous locale (invisibly).

names2 *Get names of a vector*

Description

This names getter always returns a character vector, even when an object does not have a names attribute. In this case, it returns a vector of empty names "". It also standardises missing names to "".

Usage

```
names2(x)
```

Arguments

x	A vector.

Examples

```
names2(letters)

# It also takes care of standardising missing names:
x <- set_names(1:3, c("a", NA, "b"))
names2(x)
```

---

new_cnd *Create a condition object*

Description

These constructors make it easy to create subclassed conditions. Conditions are objects that power the error system in R. They can also be used for passing messages to pre-established handlers.

Usage

```
new_cnd(.type = NULL, ..., .msg = NULL)

cnd_error(.type = NULL, ..., .msg = NULL)

cnd_warning(.type = NULL, ..., .msg = NULL)

cnd_message(.type = NULL, ..., .msg = NULL)
```

Arguments

.type	The condition subclass.
...	Named data fields stored inside the condition object. These dots are evaluated with explicit splicing.
.msg	A default message to inform the user about the condition when it is signalled.

Details

`new_cnd()` creates objects inheriting from `condition`. Conditions created with `cnd_error()`, `cnd_warning()` and `cnd_message()` inherit from `error`, `warning` or `message`.

See Also

`cnd_signal()`, `with_handlers()`.

Examples

```
# Create a condition inheriting from the s3 type "foo":
cnd <- new_cnd("foo")

# Signal the condition to potential handlers. This has no effect if no
# handler is registered to deal with conditions of type "foo":
cnd_signal(cnd)

# If a relevant handler is on the current evaluation stack, it will be
# called by cnd_signal():
with_handlers(cnd_signal(cnd), foo = exiting(function(c) "caught!"))

# Handlers can be thrown or executed inplace. See with_handlers()
# documentation for more on this.


# Note that merely signalling a condition inheriting of "error" is
# not sufficient to stop a program:
cnd_signal(cnd_error("my_error"))

# you need to use stop() to signal a critical condition that should
# terminate the program if not handled:
# stop(cnd_error("my_error"))
```

---

new_formula *Create a formula*

Description

Create a formula

Usage

```
new_formula(lhs, rhs, env = caller_env())
```

Arguments

lhs, rhs	A call, name, or atomic vector.
env	An environment.

Value

A formula object.

See Also

new_quosure()

Examples

```
new_formula(quote(a), quote(b))
new_formula(NULL, quote(b))
```

`new_function` *Create a function*

Description

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

Usage

```
new_function(args, body, env = caller_env())
```

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 alist, e.g. `alist(a = , b = 1)`
`body`	A language object representing the code inside the function. Usually this will be most easily generated with `base::quote()`
`env`	The parent environment of the function, defaults to the calling environment of `make_function`

Examples

```
f <- function(x) x + 3
g <- new_function(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))
```

ns_env *Get the namespace of a package*

Description

Namespaces are the environment where all the functions of a package live. The parent environments of namespaces are the `imports` environments, which contain all the functions imported from other packages.

Usage

```
ns_env(pkg = NULL)

ns_imports_env(pkg = NULL)

ns_env_name(pkg = NULL)
```

Arguments

pkg	The name of a package. If `NULL`, the surrounding namespace is returned, or an error is issued if not called within a namespace. If a function, the enclosure of that function is checked.

See Also

`pkg_env()`

op-definition *Definition operator*

Description

The definition operator is typically used in DSL packages like `ggvis` and `data.table`. It is exported in rlang as a alias to `~`. This makes it a quoting operator that can be shared between packages for computing on the language. Since it effectively creates formulas, it is immediately compatible with rlang's formulas and interpolation features.

Usage

```
":="()

is_definition(x)

new_definition(lhs, rhs, env = caller_env())
```

Arguments

x	An object to test.
lhs, rhs	Expressions for the LHS and RHS of the definition.
env	The evaluation environment bundled with the definition.

Examples

```
# This is useful to provide an alternative way of specifying
# arguments in DSLs:
fn <- function(...) ..1
f <- fn(arg := foo(bar) + baz)

is_formula(f)
f_lhs(f)
f_rhs(f)

# A predicate is provided to distinguish formulas from the
# colon-equals operator:
is_definition(a := b)
is_definition(a ~ b)
```

---

op-get-attr *Infix attribute accessor*

Description

Infix attribute accessor

Usage

```
x %@% name
```

Arguments

x	Object
name	Attribute name

Examples

```
factor(1:3) %@% "levels"
mtcars %@% "class"
```

op-na-default *Replace missing values*

Description

This infix function is similar to `%||%` but is vectorised and provides a default value for missing elements. It is faster than using `base::ifelse()` and does not perform type conversions.

Usage

```
x %|% y
```

Arguments

`x, y`	`y` for elements of `x` that are NA; otherwise, `x`.

See Also

op-null-default

Examples

```
c("a", "b", NA, "c") %|% "default"
```

op-null-default *Default value for* `NULL`

Description

This infix function makes it easy to replace `NULL`s with a default value. It's inspired by the way that Ruby's or operation (`||`) works.

Usage

```
x %||% y
```

Arguments

`x, y`	If `x` is NULL, will return `y`; otherwise returns `x`.

Examples

```
1 %||% 2
NULL %||% 2
```

`pairlist` *Helpers for pairlist and language nodes*

Description

Like any parse tree, R expressions are structured as trees of nodes. Each node has two components: the head and the tail (though technically there is actually a third component for argument names, see details). Due to R's lisp roots, the head of a node (or cons cell) is called the CAR and the tail is called the CDR (pronounced *car* and *cou-der*). While R's ordinary subsetting operators have builtin support for indexing into these trees and replacing elements, it is sometimes useful to manipulate the nodes more directly. This is the purpose of functions like `node_car()` and `mut_node_car()`. They are particularly useful to prototype algorithms for your C-level functions.

- `node_car()` and `mut_node_car()` access or change the head of a node.
- `node_cdr()` and `mut_node_cdr()` access or change the tail of a node.
- Variants like `node_caar()` or `mut_node_cdar()` deal with the CAR of the CAR of a node or the CDR of the CAR of a node respectively. The letters in the middle indicate the type (CAR or CDR) and order of access.
- `node_tag()` and `mut_node_tag()` access or change the tag of a node. This is meant for argument names and should only contain symbols (not strings).
- `node()` creates a new node from two components.

Usage

```
node(newcar, newcdr)

node_car(x)

node_cdr(x)

node_caar(x)

node_cadr(x)

node_cdar(x)

node_cddr(x)

mut_node_car(x, newcar)

mut_node_cdr(x, newcdr)

mut_node_caar(x, newcar)

mut_node_cadr(x, newcar)
```

```
mut_node_cdar(x, newcdr)

mut_node_cddr(x, newcdr)

node_tag(x)

mut_node_tag(x, newtag)
```

Arguments

`newcar, newcdr`	The new CAR or CDR for the node. These can be any R objects.
`x`	A language or pairlist node. Note that these functions are barebones and do not perform any type checking.
`newtag`	The new tag for the node. This should be a symbol.

Details

R has two types of nodes to represent parse trees: language nodes, which represent function calls, and pairlist nodes, which represent arguments in a function call. These are the exact same data structures with a different name. This distinction is helpful for parsing the tree: the top-level node of a function call always has *language* type while its arguments have *pairlist* type.

Note that it is risky to manipulate calls at the node level. First, the calls are changed inplace. This is unlike base R operators which create a new copy of the language tree for each modification. To make sure modifying a language object does not produce side-effects, rlang exports the `duplicate()` function to create deep copy (or optionally a shallow copy, i.e. only the top-level node is copied). The second danger is that R expects language trees to be structured as a `NULL`-terminated list. The CAR of a node is a data slot and can contain anything, including another node (which is how you form trees, as opposed to mere linked lists). On the other hand, the CDR has to be either another node, or `NULL`. If it is terminated by anything other than the `NULL` object, many R commands will crash, including functions like `str()`. It is up to you to ensure that the language list you have modified is `NULL`-terminated.

Finally, all nodes can contain metadata in the TAG slot. This is meant for argument names and R expects tags to contain a symbol (not a string).

Value

Setters like `mut_node_car()` invisibly return `x` modified in place. Getters return the requested node component.

See Also

`duplicate()` for creating copy-safe objects, `lang_head()` and `lang_tail()` as slightly higher level alternatives that check their input, and `base::pairlist()` for an easier way of creating a linked list of nodes.

Examples

```
# Changing a node component happens in place and can have side
# effects. Let's create a language object and a copy of it:
lang <- quote(foo(bar))
copy <- lang

# Using R's builtin operators to change the language tree does not
# create side effects:
copy[[2]] <- quote(baz)
copy
lang

# On the other hand, the CAR and CDR operators operate in-place. Let's
# create new objects since the previous examples triggered a copy:
lang <- quote(foo(bar))
copy <- lang

# Now we change the argument pairlist of `copy`, making sure the new
# arguments are NULL-terminated:
mut_node_cdr(copy, node(quote(BAZ), NULL))

# Or equivalently:
mut_node_cdr(copy, pairlist(quote(BAZ)))
copy

# The original object has been changed in place:
lang
```

---

parse_expr *Parse R code*

Description

These functions parse and transform text into R expressions. This is the first step to interpret or evaluate a piece of R code written by a programmer.

Usage

```
parse_expr(x)

parse_exprs(x)

parse_quosure(x, env = caller_env())

parse_quosures(x, env = caller_env())
```

Arguments

x	Text containing expressions to parse_expr for `parse_expr()` and `parse_exprs()`. Can also be an R connection, for instance to a file. If the supplied connection is not open, it will be automatically closed and destroyed.
env	The environment for the formulas. Defaults to the context in which the parse_expr function was called. Can be any object with a `as_env()` method.

Details

`parse_expr()` returns one expression. If the text contains more than one expression (separated by colons or new lines), an error is issued. On the other hand `parse_exprs()` can handle multiple expressions. It always returns a list of expressions (compare to `base::parse()` which returns an base::expression vector). All functions also support R connections.

The versions prefixed with `f_` return expressions quoted in formulas rather than raw expressions.

Value

`parse_expr()` returns a formula, `parse_exprs()` returns a list of formulas.

See Also

`base::parse()`

Examples

```
# parse_expr() can parse_expr any R expression:
parse_expr("mtcars %>% dplyr::mutate(cyl_prime = cyl / sd(cyl))")

# A string can contain several expressions separated by ; or \n
parse_exprs("NULL; list()\n foo(bar)")

# The versions suffixed with _f return formulas:
parse_quosure("foo %>% bar()")
parse_quosures("1; 2; mtcars")

# The env argument is passed to as_env(). It can be e.g. a string
# representing a scoped package environment:
parse_quosure("identity(letters)", env = empty_env())
parse_quosures("identity(letters); mtcars", env = "base")


# You can also parse source files by passing a R connection. Let's
# create a file containing R code:
path <- tempfile("my-file.R")
cat("1; 2; mtcars", file = path)

# We can now parse it by supplying a connection:
parse_exprs(file(path))
```

prepend *Prepend a vector*

Description

This is a companion to base::append() to help merging two lists or atomic vectors. prepend() is a clearer semantic signal than c() that a vector is to be merged at the beginning of another, especially in a pipe chain.

Usage

```
prepend(x, values, before = 1)
```

Arguments

x	the vector to be modified.
values	to be included in the modified vector.
before	a subscript, before which the values are to be appended.

Value

A merged vector.

Examples

```
x <- as.list(1:3)

append(x, "a")
prepend(x, "a")
prepend(x, list("a", "b"), before = 3)
```

prim_name *Name of a primitive function*

Description

Name of a primitive function

Usage

```
prim_name(prim)
```

Arguments

prim	A primitive function such as base::c().

quasiquotation *Quasiquotation of an expression*

Description

Quasiquotation is the mechanism that makes it possible to program flexibly with tidyeval grammars like dplyr. It is enabled in all tidyeval functions, the most fundamental of which are `quo()` and `expr()`.

Quasiquotation is the combination of quoting an expression while allowing immediate evaluation (unquoting) of part of that expression. We provide both syntactic operators and functional forms for unquoting.

- `UQ()` and the `!!` operator unquote their argument. It gets evaluated immediately in the surrounding context.
- `UQE()` is like `UQ()` but retrieves the expression of quosureish objects. It is a shortcut for `!! get_expr(x)`. Use this with care: it is potentially unsafe to discard the environment of the quosure.
- `UQS()` and the `!!!` operators unquote and splice their argument. The argument should evaluate to a vector or an expression. Each component of the vector is embedded as its own argument in the surrounding call. If the vector is named, the names are used as argument names.

Usage

```
UQ(x)

UQE(x)

"!!"(x)

UQS(x)
```

Arguments

x	An expression to unquote.

Theory

Formally, `quo()` and `expr()` are quasiquote functions, `UQ()` is the unquote operator, and `UQS()` is the unquote splice operator. These terms have a rich history in Lisp languages, and live on in modern languages like Julia and Racket.

Examples

```
# Quasiquotation functions act like base::quote()
quote(foo(bar))
expr(foo(bar))
quo(foo(bar))
```

```
# In addition, they support unquoting:
expr(foo(UQ(1 + 2)))
expr(foo(!! 1 + 2))
quo(foo(!! 1 + 2))

# The !! operator is a handy syntactic shortcut for unquoting with
# UQ().  However you need to be a bit careful with operator
# precedence. All arithmetic and comparison operators bind more
# tightly than `!`:
quo(1 +  !! (1 + 2 + 3) + 10)

# For this reason you should always wrap the unquoted expression
# with parentheses when operators are involved:
quo(1 + (!! 1 + 2 + 3) + 10)

# Or you can use the explicit unquote function:
quo(1 + UQ(1 + 2 + 3) + 10)


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

# You can combine the two
var <- quote(xyz)
extra_args <- list(trim = 0.9, na.rm = TRUE)
quo(mean(UQ(var) , UQS(extra_args)))


# Unquoting is especially useful for transforming successively a
# captured expression:
quo <- quo(foo(bar))
quo <- quo(inner(!! quo, arg1))
quo <- quo(outer(!! quo, !!! syms(letters[1:3])))
quo

# Since we are building the expression in the same environment, you
# can also start with raw expressions and create a quosure in the
# very last step to record the dynamic environment:
expr <- expr(foo(bar))
expr <- expr(inner(!! expr, arg1))
quo <- quo(outer(!! expr, !!! syms(letters[1:3])))
quo
```

---

quo-predicates *Is a quosure quoting a symbolic, missing or NULL object?*

Description

These functions examine the expression of a quosure with a predicate.

Usage

```
quo_is_missing(quo)

quo_is_symbol(quo)

quo_is_lang(quo)

quo_is_symbolic(quo)

quo_is_null(quo)
```

Arguments

quo	A quosure.

Empty quosures

When missing arguments are captured as quosures, either through `enquo()` or `quos()`, they are returned as an empty quosure. These quosures contain the missing argument and typically have the empty environment as enclosure.

Examples

```
quo_is_symbol(quo(sym))
quo_is_symbol(quo(foo(bar)))

# You can create empty quosures by calling quo() without input:
quo <- quo()
quo_is_missing(quo)
is_missing(f_rhs(quo))
```

---

quosure *Create quosures*

Description

Quosures are quoted expressions that keep track of an environment (just like closurefunctions). They are implemented as a subclass of one-sided formulas. They are an essential piece of the tidy evaluation framework.

- `quo()` quotes its input (i.e. captures R code without evaluation), captures the current environment, and bundles them in a quosure.

- `enquo()` takes a symbol referring to a function argument, quotes the R code that was supplied to this argument, captures the environment where the function was called (and thus where the R code was typed), and bundles them in a quosure.
- `quos()` is a bit different to other functions as it returns a list of quosures. You can supply several expressions directly, e.g. `quos(foo, bar)`, but more importantly you can also supply dots: `quos(...)`. In the latter case, expressions forwarded through dots are captured and transformed to quosures. The environments bundled in those quosures are the ones where the code was supplied as arguments, even if the dots were forwarded multiple times across several function calls.
- `new_quosure()` is the only constructor that takes its arguments by value. It lets you create a quosure from an expression and an environment.

Usage

```
quo(expr)

new_quosure(expr, env = caller_env())

enquo(arg)
```

Arguments

`expr`	An expression.
`env`	An environment specifying the lexical enclosure of the quosure.
`arg`	A symbol referring to an argument. The expression supplied to that argument will be captured unevaluated.

Value

A formula whose right-hand side contains the quoted expression supplied as argument.

Role of quosures for tidy evaluation

Quosures play an essential role thanks to these features:

- They allow consistent scoping of quoted expressions by recording an expression along with its local environment.
- `quo()`, `quos()` and `enquo()` all support quasiquotation. By unquoting other quosures, you can safely combine expressions even when they come from different contexts. You can also unquote values and raw expressions depending on your needs.
- Unlike formulas, quosures self-evaluate (see `eval_tidy()`) within their own environment, which is why you can unquote a quosure inside another quosure and evaluate it like you've unquoted a raw expression.

See the programming withdplyr vignette for practical examples. For developers, the tidyevaluation vignette provides an overview of this approach. The quasiquotation page goes in detail over the unquoting and splicing operators.

See Also

expr() for quoting a raw expression with quasiquotation. The quasiquotation page goes over unquoting and splicing.

Examples

```
# quo() is a quotation function just like expr() and quote():
expr(mean(1:10 * 2))
quo(mean(1:10 * 2))

# It supports quasiquotation and allows unquoting (evaluating
# immediately) part of the quoted expression:
quo(mean(!! 1:10 * 2))

# What makes quo() often safer to use than quote() and expr() is
# that it keeps track of the contextual environment. This is
# especially important if you're referring to local variables in
# the expression:
var <- "foo"
quo <- quo(var)
quo

# Here `quo` quotes `var`. Let's check that it also captures the
# environment where that symbol is defined:
identical(get_env(quo), get_env())
env_has(quo, "var")


# Keeping track of the environment is important when you quote an
# expression in a context (that is, a particular function frame)
# and pass it around to other functions (which will be run in their
# own evaluation frame):
fn <- function() {
  foobar <- 10
  quo(foobar * 2)
}
quo <- fn()
quo

# `foobar` is not defined here but was defined in `fn()`'s
# evaluation frame. However, the quosure keeps track of that frame
# and is safe to evaluate:
eval_tidy(quo)


# Like other formulas, quosures are normally self-quoting under
# evaluation:
eval(~var)
eval(quo(var))

# But eval_tidy() evaluates expressions in a special environment
# (called the overscope) where they become promises. They
```

```
# self-evaluate under evaluation:
eval_tidy(~var)
eval_tidy(quo(var))

# Note that it's perfectly fine to unquote quosures within
# quosures, as long as you evaluate with eval_tidy():
quo <- quo(letters)
quo <- quo(toupper(!! quo))
quo
eval_tidy(quo)


# Quoting as a quosure is necessary to preserve scope information
# and make sure objects are looked up in the right place. However,
# there are situations where it can get in the way. This is the
# case when you deal with non-tidy NSE functions that do not
# understand formulas. You can inline the RHS of a formula in a
# call thanks to the UQE() operator:
nse_function <- function(arg) substitute(arg)
var <- locally(quo(foo(bar)))
quo(nse_function(UQ(var)))
quo(nse_function(UQE(var)))

# This is equivalent to unquoting and taking the RHS:
quo(nse_function(!! get_expr(var)))

# One of the most important old-style NSE function is the dollar
# operator. You need to use UQE() for subsetting with dollar:
var <- quo(cyl)
quo(mtcars$UQE(var))

# `!!`() is also treated as a shortcut. It is meant for situations
# where the bang operator would not parse, such as subsetting with
# $. Since that's its main purpose, we've made it a shortcut for
# UQE() rather than UQ():
var <- quo(cyl)
quo(mtcars$`!!`(var))


# When a quosure is printed in the console, the brackets indicate
# if the enclosure is the global environment or a local one:
locally(quo(foo))

# Literals are enquosed with the empty environment because they can
# be evaluated anywhere. The brackets indicate "empty":
quo(10L)

# To differentiate local environments, use str(). It prints the
# machine address of the environment:
quo1 <- locally(quo(foo))
quo2 <- locally(quo(foo))
quo1; quo2
str(quo1); str(quo2)
```

```
# You can also see this address by printing the environment at the
# console:
get_env(quo1)
get_env(quo2)


# new_quosure() takes by value an expression that is already quoted:
expr <- quote(mtcars)
env <- as_env("datasets")
quo <- new_quosure(expr, env)
quo
eval_tidy(quo)
```

---

quo_expr *Splice a quosure and format it into string or label*

Description

quo_expr() flattens all quosures within an expression. I.e., it turns ~foo(~bar(), ~baz) to foo(bar(), baz). quo_text() and quo_label() are equivalent to f_text(), expr_label(), etc, but they first splice their argument using quo_expr(). quo_name() transforms a quoted symbol to a string. It adds a bit more intent and type checking than simply calling quo_text() on the quoted symbol (which will work but won't return an error if not a symbol).

Usage

```
quo_expr(quo, warn = FALSE)

quo_label(quo)

quo_text(quo, width = 60L, nlines = Inf)

quo_name(quo)
```

Arguments

quo	A quosure or expression.
warn	Whether to warn if the quosure contains other quosures (those will be collapsed).
width	Width of each line.
nlines	Maximum number of lines to extract.

See Also

expr_label(), f_label()

Examples

```
quo <- quo(foo(!! quo(bar)))
quo

# quo_expr() unwraps all quosures and returns a raw expression:
quo_expr(quo)

# This is used by quo_text() and quo_label():
quo_text(quo)

# Compare to the unwrapped expression:
expr_text(quo)

# quo_name() is helpful when you need really short labels:
quo_name(quo(sym))
quo_name(quo(!! sym))
```

`restarting` *Create a restarting handler*

Description

This constructor automates the common task of creating an `inplace()` handler that invokes a restart.

Usage

```
restarting(.restart, ..., .fields = NULL)
```

Arguments

`.restart`	The name of a restart.
`...`	Additional arguments passed on the restart function. These arguments are evaluated only once and immediately, when creating the restarting handler. Furthermore, they are evaluated with explicit splicing.
`.fields`	A character vector specifying the fields of the condition that should be passed as arguments to the restart. If named, the names (except empty names `""`) are used as argument names for calling the restart function. Otherwise the the fields themselves are used as argument names.

Details

Jumping to a restart point from an inplace handler has two effects. First, the control flow jumps to wherever the restart was established, and the restart function is called (with `...`, or `.fields` as arguments). Execution resumes from the `with_restarts()` call. Secondly, the transfer of the control flow out of the function that signalled the condition means that the handler has dealt with the condition. Thus the condition will not be passed on to other potential handlers established on the stack.

See Also

inplace() and exiting().

Examples

```
# This is a restart that takes a data frame and names as arguments
rst_bar <- function(df, nms) {
  stats::setNames(df, nms)
}

# This restart is simpler and does not take arguments
rst_baz <- function() "baz"

# Signalling a condition parameterised with a data frame
fn <- function() {
  with_restarts(cnd_signal("foo", foo_field = mtcars),
    rst_bar = rst_bar,
    rst_baz = rst_baz
  )
}

# Creating a restarting handler that passes arguments `nms` and
# `df`, the latter taken from a data field of the condition object
restart_bar <- restarting("rst_bar",
  nms = LETTERS[1:11], .fields = c(df = "foo_field")
)

# The restarting handlers jumps to `rst_bar` when `foo` is signalled:
with_handlers(fn(), foo = restart_bar)

# The restarting() constructor is especially nice to use with
# restarts that do not need arguments:
with_handlers(fn(), foo = restarting("rst_baz"))
```

---

return_from *Jump to or from a frame*

Description

While base::return() can only return from the current local frame, these two functions will return from any frame on the current evaluation stack, between the global and the currently active context. They provide a way of performing arbitrary non-local jumps out of the function currently under evaluation.

Usage

```
return_from(frame, value = NULL)

return_to(frame, value = NULL)
```

Arguments

frame	An environment, a frame object, or any object with an get_env() method. The environment should be an evaluation environment currently on the stack.
value	The return value.

Details

return_from() will jump out of frame. return_to() is a bit trickier. It will jump out of the frame located just before frame in the evaluation stack, so that control flow ends up in frame, at the location where the previous frame was called from.

These functions should only be used rarely. These sort of non-local gotos can be hard to reason about in casual code, though they can sometimes be useful. Also, consider to use the condition system to perform non-local jumps.

Examples

```
# Passing fn() evaluation frame to g():
fn <- function() {
  val <- g(get_env())
  cat("g returned:", val, "\n")
  "normal return"
}
g <- function(env) h(env)

# Here we return from fn() with a new return value:
h <- function(env) return_from(env, "early return")
fn()

# Here we return to fn(). The call stack unwinds until the last frame
# called by fn(), which is g() in that case.
h <- function(env) return_to(env, "early return")
fn()
```

---

rst_abort *Jump to the abort restart*

Description

The abort restart is the only restart that is established at top level. It is used by R as a top-level target, most notably when an error is issued (see abort()) that no handler is able to deal with (see with_handlers()).

Usage

```
rst_abort()
```

See Also

rst_jump(), abort() and cnd_abort().

Examples

```
# The `abort` restart is a bit special in that it is always
# registered in a R session. You will always find it on the restart
# stack because it is established at top level:
rst_list()

# You can use the `above` restart to jump to top level without
# signalling an error:
## Not run:
fn <- function() {
  cat("aborting...\n")
  rst_abort()
  cat("This is never called\n")
}
{
  fn()
  cat("This is never called\n")
}

## End(Not run)

# The `above` restart is the target that R uses to jump to top
# level when critical errors are signalled:
## Not run:
{
  abort("error")
  cat("This is never called\n")
}

## End(Not run)

# If another `abort` restart is specified, errors are signalled as
# usual but then control flow resumes with from the new restart:
## Not run:
out <- NULL
{
  out <- with_restarts(abort("error"), abort = function() "restart!")
  cat("This is called\n")
}
cat("`out` has now become:", out, "\n")

## End(Not run)
```

---

rst_list *Restarts utilities*

Description

Restarts are named jumping points established by `with_restarts()`. `rst_list()` returns the names of all restarts currently established. `rst_exists()` checks if a given restart is established. `rst_jump()` stops execution of the current function and jumps to a restart point. If the restart does not exist, an error is thrown. `rst_maybe_jump()` first checks that a restart exists before jumping.

Usage

```
rst_list()

rst_exists(.restart)

rst_jump(.restart, ...)

rst_maybe_jump(.restart, ...)
```

Arguments

`.restart`	The name of a restart.
`...`	Arguments passed on to the restart function. These dots are evaluated with explicit splicing.

See Also

`with_restarts()`, `rst_muffle()`.

---

rst_muffle *Jump to a muffling restart*

Description

Muffle restarts are established at the same location as where a condition is signalled. They are useful for two non-exclusive purposes: muffling signalling functions and muffling conditions. In the first case, `rst_muffle()` prevents any further side effects of a signalling function (a warning or message from being displayed, an aborting jump to top level, etc). In the second case, the muffling jump prevents a condition from being passed on to other handlers. In both cases, execution resumes normally from the point where the condition was signalled.

Usage

```
rst_muffle(c)
```

Arguments

`c`	A condition to muffle.

See Also

The `muffle` argument of `inplace()`, and the `mufflable` argument of `cnd_signal()`.

Examples

```
side_effect <- function() cat("side effect!\n")
handler <- inplace(function(c) side_effect())

# A muffling handler is an inplace handler that jumps to a muffle
# restart:
muffling_handler <- inplace(function(c) {
  side_effect()
  rst_muffle(c)
})

# You can also create a muffling handler simply by setting
# muffle = TRUE:
muffling_handler <- inplace(function(c) side_effect(), muffle = TRUE)

# You can then muffle the signalling function:
fn <- function(signal, msg) {
  signal(msg)
  "normal return value"
}
with_handlers(fn(message, "some message"), message = handler)
with_handlers(fn(message, "some message"), message = muffling_handler)
with_handlers(fn(warning, "some warning"), warning = muffling_handler)

# Note that exiting handlers are thrown to the establishing point
# before being executed. At that point, the restart (established
# within the signalling function) does not exist anymore:
## Not run:
with_handlers(fn(warning, "some warning"),
  warning = exiting(function(c) rst_muffle(c)))

## End(Not run)


# Another use case for muffle restarts is to muffle conditions
# themselves. That is, to prevent other condition handlers from
# being called:
undesirable_handler <- inplace(function(c) cat("please don't call me\n"))

with_handlers(foo = undesirable_handler,
  with_handlers(foo = muffling_handler, {
    cnd_signal("foo", mufflable = TRUE)
    "return value"
  }))

# See the `mufflable` argument of cnd_signal() for more on this point
```

scalar-type-predicates
Scalar type predicates

Description

These predicates check for a given type and whether the vector is "scalar", that is, of length 1.

Usage

```
is_scalar_list(x)

is_scalar_atomic(x)

is_scalar_vector(x)

is_scalar_integer(x)

is_scalar_double(x)

is_scalar_character(x, encoding = NULL)

is_scalar_logical(x)

is_scalar_raw(x)

is_string(x, encoding = NULL)

is_scalar_bytes(x)
```

Arguments

x	object to be tested.
encoding	Expected encoding of a string or character vector. One of `UTF-8`, `latin1`, or `unknown`.

See Also

type-predicates, bare-type-predicates

scoped_env *Scoped environments*

Description

Scoped environments are named environments which form a parent-child hierarchy called the search path. They define what objects you can see (are in scope) from your workspace. They typically are package environments, i.e. special environments containing all exported functions from a package (and whose parent environment is the package namespace, which also contains unexported functions). Package environments are attached to the search path with `base::library()`. Note however that any environment can be attached to the search path, for example with the unrecommended `base::attach()` base function which transforms vectors to scoped environments.

- You can list all scoped environments with `scoped_names()`. Unlike `base::search()`, it also mentions the empty environment that terminates the search path (it is given the name `"NULL"`).
- `scoped_envs()` returns all environments on the search path, including the empty environment.
- `pkg_env()` takes a package name and returns the scoped environment of packages if they are attached to the search path, and throws an error otherwise.
- `is_scoped()` allows you to check whether a named environment is on the search path.

Usage

```
scoped_env(nm)

pkg_env(pkg)

pkg_env_name(pkg)

scoped_names()

scoped_envs()

is_scoped(nm)

base_env()

global_env()
```

Arguments

`nm`	The name of an environment attached to the search path. Call `base::search()` to see what is currently on the path.
`pkg`	The name of a package.

Search path

The search path is a chain of scoped environments where newly attached environments are the childs of earlier ones. However, the global environment, where everything you define at top-level ends up, is pinned as the head of that linked chain. Likewise, the base package environment is pinned as the tail of the chain. You can retrieve those environments with `global_env()` and `base_env()` respectively. The global environment is also the environment of the very first evaluation frame on the stack, see `global_frame()` and `ctxt_stack()`.

Examples

```
# List the names of scoped environments:
nms <- scoped_names()
nms

# The global environment is always the first in the chain:
scoped_env(nms[[1]])

# And the scoped environment of the base package is always the last:
scoped_env(nms[[length(nms)]])

# These two environments have their own shortcuts:
global_env()
base_env()

# Packages appear in the search path with a special name. Use
# pkg_env_name() to create that name:
pkg_env_name("rlang")
scoped_env(pkg_env_name("rlang"))

# Alternatively, get the scoped environment of a package with
# pkg_env():
pkg_env("utils")
```

---

seq2 *Increasing sequence of integers in an interval*

Description

These helpers take two endpoints and return the sequence of all integers within that interval. For `seq2_along()`, the upper endpoint is taken from the length of a vector. Unlike `base::seq()`, they return an empty vector if the starting point is a larger integer than the end point.

Usage

```
seq2(from, to)

seq2_along(from, x)
```

Arguments

from	The starting point of the sequence.
to	The end point.
x	A vector whose length is the end point.

Value

An integer vector containing a strictly increasing sequence.

Examples

```
seq2(2, 10)
seq2(10, 2)
seq(10, 2)

seq2_along(10, letters)
```

---

set_attrs *Add attributes to an object*

Description

`set_attrs()` adds, changes, or zaps attributes of objects. Pass a single unnamed `NULL` as argument to zap all attributes. For uncopyable types, use `mut_attrs()`.

Usage

```
set_attrs(.x, ...)

mut_attrs(.x, ...)
```

Arguments

.x	An object to decorate with attributes.
...	A list of named attributes. These have explicit splicing semantics. Pass a single unnamed `NULL` to zap all attributes from `.x`.

Details

Unlike `structure()`, these setters have no special handling of internal attributes names like `.Dim`, `.Dimnames` or `.Names`.

Value

`set_attrs()` returns a modified shallow copy of `.x`. `mut_attrs()` invisibly returns the original `.x` modified in place.

Examples

```
set_attrs(letters, names = 1:26, class = "my_chr")

# Splice a list of attributes:
attrs <- list(attr = "attr", names = 1:26, class = "my_chr")
obj <- set_attrs(letters, splice(attrs))
obj

# Zap attributes by passing a single unnamed NULL argument:
set_attrs(obj, NULL)
set_attrs(obj, !!! list(NULL))

# Note that set_attrs() never modifies objects in place:
obj

# For uncopyable types, mut_attrs() lets you modify in place:
env <- env()
mut_attrs(env, foo = "bar")
env
```

---

set_chr_encoding *Set encoding of a string or character vector*

Description

R has specific support for UTF-8 and latin1 encoded strings. This mostly matters for internal conversions. Thanks to this support, you can reencode strings to UTF-8 or latin1 for internal processing, and return these strings without having to convert them back to the native encoding. However, it is important to make sure the encoding mark has not been lost in the process, otherwise the output will be treated as if encoded according to the current locale (see `mut_utf8_locale()` for documentation about locale codesets), which is not appropriate if it does not coincide with the actual encoding. In those situations, you can use these functions to ensure an encoding mark in your strings.

Usage

```
set_chr_encoding(x, encoding = c("unknown", "UTF-8", "latin1", "bytes"))

chr_encoding(x)

set_str_encoding(x, encoding = c("unknown", "UTF-8", "latin1", "bytes"))

str_encoding(x)
```

Arguments

x	A string or character vector.
encoding	Either an encoding specially handled by R (`"UTF-8"` or `"latin1"`), `"bytes"` to inhibit all encoding conversions, or `"unknown"` if the string should be treated as encoded in the current locale codeset.

See Also

mut_utf8_locale() about the effects of the locale, and as_utf8_string() about encoding conversion.

Examples

```
# Encoding marks are always ignored on ASCII strings:
str_encoding(set_str_encoding("cafe", "UTF-8"))

# You can specify the encoding of strings containing non-ASCII
# characters:
cafe <- string(c(0x63, 0x61, 0x66, 0xC3, 0xE9))
str_encoding(cafe)
str_encoding(set_str_encoding(cafe, "UTF-8"))


# It is important to consistently mark the encoding of strings
# because R and other packages perform internal string conversions
# all the time. Here is an example with the names attribute:
latin1 <- string(c(0x63, 0x61, 0x66, 0xE9), "latin1")
latin1 <- set_names(latin1)

# The names attribute is encoded in latin1 as we would expect:
str_encoding(names(latin1))

# However the names are converted to UTF-8 by the c() function:
str_encoding(names(c(latin1)))
as_bytes(names(c(latin1)))

# Bad things happen when the encoding marker is lost and R performs
# a conversion. R will assume that the string is encoded according
# to the current locale:
## Not run:
bad <- set_names(set_str_encoding(latin1, "unknown"))
mut_utf8_locale()

str_encoding(names(c(bad)))
as_bytes(names(c(bad)))

## End(Not run)
```

---

set_expr *Set and get an expression*

Description

These helpers are useful to make your function work generically with quosures and raw expressions. First call `get_expr()` to extract an expression. Once you're done processing the expression, call `set_expr()` on the original object to update the expression. You can return the result of

`set_expr()`, either a formula or an expression depending on the input type. Note that `set_expr()` does not change its input, it creates a new object.

Usage

```
set_expr(x, value)

get_expr(x, default = x)
```

Arguments

`x`	An expression or one-sided formula. In addition, `set_expr()` accept frames.
`value`	An updated expression.
`default`	A default expression to return when `x` is not an expression wrapper. Defaults to `x` itself.

Value

The updated original input for `set_expr()`. A raw expression for `get_expr()`.

Examples

```
f <- ~foo(bar)
e <- quote(foo(bar))
frame <- identity(identity(ctxt_frame()))

get_expr(f)
get_expr(e)
get_expr(frame)

set_expr(f, quote(baz))
set_expr(e, quote(baz))
```

`set_names` *Set names of a vector*

Description

This is equivalent to `stats::setNames()`, with more features and stricter argument checking.

Usage

```
set_names(x, nm = x, ...)
```

Arguments

x	Vector to name.
nm, ...	Vector of names, the same length as x. You can specify names in the following ways: • If you do nothing, x will be named with itself. • If x already has names, you can provide a function or formula to transform the existing names. In that case, ... is passed to the function. • If nm is NULL, the names are removed (if present). • In all other cases, nm and ... are passed to chr(). This gives implicit splicing semantics: you can pass character vectors or list of character vectors indistinctly.

Examples

```
set_names(1:4, c("a", "b", "c", "d"))
set_names(1:4, letters[1:4])
set_names(1:4, "a", "b", "c", "d")

# If the second argument is ommitted a vector is named with itself
set_names(letters[1:5])

# Alternatively you can supply a function
set_names(1:10, ~ letters[seq_along(.)])
set_names(head(mtcars), toupper)

# `...` is passed to the function:
set_names(head(mtcars), paste0, "_foo")
```

---

splice *Splice a list within a vector*

Description

This adjective signals to functions taking dots that x should be spliced in a surrounding vector. Examples of functions that support such explicit splicing are ll(), chr(), etc. Generally, any functions taking dots with dots_list() or dots_splice() supports splicing.

Usage

```
splice(x)

is_spliced(x)

is_spliced_bare(x)
```

Arguments

x	A list to splice.

Details

Note that all functions supporting dots splicing also support the syntactic operator `!!!`. For tidy capture and tidy evaluation, this operator directly manipulates the calls (see `quo()` and quasiquotation). However manipulating the call is not appropriate when taking dots by value rather than by expression, because it is slow and the dots might contain large lists of data. For this reason we splice values rather than expressions when dots are not captured by expression. We do it in two steps: first mark the objects to be spliced, then splice the objects with `flatten()`.

See Also

vector-construction

Examples

```
x <- list("a")

# It makes sense for ll() to accept lists literally, so it doesn't
# automatically splice them:
ll(x)

# But you can splice lists explicitly:
y <- splice(x)
ll(y)

# Or with the syntactic shortcut:
ll(!!! x)
```

---

stack *Call stack information*

Description

The `eval_` and `call_` families of functions provide a replacement for the base R functions prefixed with `sys.` (which are all about the context stack), as well as for `parent.frame()` (which is the only base R function for querying the call stack). The context stack includes all R-level evaluation contexts. It is linear in terms of execution history but due to lazy evaluation it is potentially nonlinear in terms of call history. The call stack history, on the other hand, is homogenous.

Usage

```
global_frame()

current_frame()
```

```
ctxt_frame(n = 1)

call_frame(n = 1, clean = TRUE)

ctxt_depth()

call_depth()

ctxt_stack(n = NULL, trim = 0)

call_stack(n = NULL, clean = TRUE)
```

Arguments

n	The number of frames to go back in the stack.
clean	Whether to post-process the call stack to clean non-standard frames. If TRUE, suboptimal call-stack entries by base::eval() will be cleaned up: the duplicate frame created by eval() is eliminated.
trim	The number of layers of intervening frames to trim off the stack. See stack_trim() and examples.

Details

ctxt_frame() and call_frame() return a frame object containing the following fields: expr and env (call expression and evaluation environment), pos and caller_pos (position of current frame in the context stack and position of the caller), and fun (function of the current frame). ctxt_stack() and call_stack() return a list of all context or call frames on the stack. Finally, ctxt_depth() and call_depth() report the current context position or the number of calling frames on the stack.

The base R functions take two sorts of arguments to indicate which frame to query: which and n. The n argument is straightforward: it's the number of frames to go down the stack, with n = 1 referring to the current context. The which argument is more complicated and changes meaning for values lower than

1. For the sake of consistency, the lazyeval functions all take the same kind of argument n. This argument has a single meaning (the number of frames to go down the stack) and cannot be lower than 1.

Note finally that parent.frame(1) corresponds to call_frame(2)$env, as n = 1 always refers to the current frame. This makes the _frame() and _stack() functions consistent: ctxt_frame(2) is the same as ctxt_stack()[[2]]. Also, ctxt_depth() returns one more frame than [base::sys.nframe()] because it counts the global frame. That is consistent with the _stack() functions which return the global frame as well. This way, call_stack(call_depth()) is the same as global_frame().

[[2]: R:[2 [base::sys.nframe()]: R:base::sys.nframe()

Examples

```
# Expressions within arguments count as contexts
identity(identity(ctxt_depth())) # returns 2
```

```
# But they are not part of the call stack because arguments are
# evaluated within the calling function (or the global environment
# if called at top level)
identity(identity(call_depth())) # returns 0

# The context stacks includes all intervening execution frames. The
# call stack doesn't:
f <- function(x) identity(x)
f(f(ctxt_stack()))
f(f(call_stack()))

g <- function(cmd) cmd()
f(g(ctxt_stack))
f(g(call_stack))

# The lazyeval _stack() functions return a list of frame
# objects. Use purrr::transpose() or index a field with
# purrr::map()'s to extract a particular field from a stack:

# stack <- f(f(call_stack()))
# purrr::map(stack, "env")
# purrr::transpose(stack)$expr

# current_frame() is an alias for ctxt_frame(1)
fn <- function() list(current = current_frame(), first = ctxt_frame(1))
fn()

# While current_frame() is the top of the stack, global_frame() is
# the bottom:
fn <- function() {
  n <- ctxt_depth()
  ctxt_frame(n)
}
identical(fn(), global_frame())


# ctxt_stack() returns a stack with all intervening frames. You can
# trim layers of intervening frames with the trim argument:
identity(identity(ctxt_stack()))
identity(identity(ctxt_stack(trim = 1)))

# ctxt_stack() is called within fn() with intervening frames:
fn <- function(trim) identity(identity(ctxt_stack(trim = trim)))
fn(0)

# We can trim the first layer of those:
fn(1)

# The outside intervening frames (at the fn() call site) are still
# returned, but can be trimmed as well:
identity(identity(fn(1)))
identity(identity(fn(2)))
```

```
g <- function(trim) identity(identity(fn(trim)))
g(2)
g(3)
```

`stack_trim` *Trim top call layers from the evaluation stack*

Description

`ctxt_stack()` can be tricky to use in real code because all intervening frames are returned with the stack, including those at `ctxt_stack()` own call site. `stack_trim()` makes it easy to remove layers of intervening calls.

Usage

```
stack_trim(stack, n = 1)
```

Arguments

`stack`	An evaluation stack.
`n`	The number of call frames (not eval frames) to trim off the top of the stack. In other words, the number of layers of intervening frames to trim.

Examples

```
# Intervening frames appear on the evaluation stack:
identity(identity(ctxt_stack()))

# stack_trim() will trim the first n layers of calls:
stack_trim(identity(identity(ctxt_stack())))

# Note that it also takes care of calls intervening at its own call
# site:
identity(identity(
  stack_trim(identity(identity(ctxt_stack())))
))

# It is especially useful when used within a function that needs to
# inspect the evaluation stack but should nonetheless be callable
# within nested calls without side effects:
stack_util <- function() {
  # n = 2 means that two layers of intervening calls should be
  # removed: The layer at ctxt_stack()'s call site (including the
  # stack_trim() call), and the layer at stack_util()'s call.
  stack <- stack_trim(ctxt_stack(), n = 2)
  stack
}
user_fn <- function() {
  # A user calls your stack utility with intervening frames:
```

```
  identity(identity(stack_util()))
}
# These intervening frames won't appear in the evaluation stack
identity(user_fn())
```

`string` *Create a string*

Description

These base-type constructors allow more control over the creation of strings in R. They take character vectors or string-like objects (integerish or raw vectors), and optionally set the encoding. The string version checks that the input contains a scalar string.

Usage

```
string(x, encoding = NULL)
```

Arguments

x	A character vector or a vector or list of string-like objects.
encoding	If non-null, passed to `set_chr_encoding()` to add an encoding mark. This is only declarative, no encoding conversion is performed.

See Also

`set_chr_encoding()` for more information about encodings in R.

Examples

```
# As everywhere in R, you can specify a string with Unicode
# escapes. The characters corresponding to Unicode codepoints will
# be encoded in UTF-8, and the string will be marked as UTF-8
# automatically:
cafe <- string("caf\uE9")
str_encoding(cafe)
as_bytes(cafe)

# In addition, string() provides useful conversions to let
# programmers control how the string is represented in memory. For
# encodings other than UTF-8, you'll need to supply the bytes in
# hexadecimal form. If it is a latin1 encoding, you can mark the
# string explicitly:
cafe_latin1 <- string(c(0x63, 0x61, 0x66, 0xE9), "latin1")
str_encoding(cafe_latin1)
as_bytes(cafe_latin1)
```

`switch_lang` *Dispatch on call type*

Description

`switch_lang()` dispatches clauses based on the subtype of call, as determined by `lang_type_of()`. The subtypes are based on the type of call head (see details).

Usage

```
switch_lang(.x, ...)

coerce_lang(.x, .to, ...)

lang_type_of(x)
```

Arguments

`.x, x`	A language object (a call). If a formula quote, the RHS is extracted first.
`...`	Named clauses. The names should be types as returned by `lang_type_of()`.
`.to`	This is useful when you switchpatch within a coercing function. If supplied, this should be a string indicating the target type. A catch-all clause is then added to signal an error stating the conversion failure. This type is prettified unless `.to` inherits from the S3 class `"AsIs"` (see `base::I()`).

Details

Calls (objects of type `language`) do not necessarily call a named function. They can also call an anonymous function or the result of some other expression. The language subtypes are organised around the kind of object being called:

- For regular calls to named function, `switch_lang()` returns "named".
- Sometimes the function being called is the result of another function call, e.g. `foo()()`, or the result of another subsetting call, e.g. `foo$bar()` or `foo@bar()`. In this case, the call head is not a symbol, it is another call (e.g. to the infix functions `$` or `@`). The call subtype is said to be "recursive".
- A special subset of recursive calls are namespaced calls like `foo::bar()`. `switch_lang()` returns "namespaced" for these calls. It is generally a good idea if your function treats `bar()` and `foo::bar()` similarly.
- Finally, it is possible to have a literal (see `is_expr()` for a definition of literals) as call head. In most cases, this will be a function inlined in the call (this is sometimes an expedient way of dealing with scoping issues). For calls with a literal node head, `switch_lang()` returns "inlined". Note that if a call head contains a literal that is not function, something went wrong and using that object will probably make R crash. `switch_lang()` issues an error in this case.

The reason we use the term *node head* is because calls are structured as tree objects. This makes sense because the best representation for language code is a parse tree, with the tree hierarchy determined by the order of operations. See pairlist for more on this.

Examples

```
# Named calls:
lang_type_of(~foo())

# Recursive calls:
lang_type_of(~foo$bar())
lang_type_of(~foo()())

# Namespaced calls:
lang_type_of(~base::list())

# For an inlined call, let's inline a function in the head node:
call <- quote(foo(letters))
call[[1]] <- base::toupper

call
lang_type_of(call)
```

---

switch_type *Dispatch on base types*

Description

`switch_type()` is equivalent to `switch(type_of(x, ...))`, while `switch_class()` switchpatches based on `class(x)`. The `coerce_` versions are intended for type conversion and provide a standard error message when conversion fails.

Usage

```
switch_type(.x, ...)

coerce_type(.x, .to, ...)

switch_class(.x, ...)

coerce_class(.x, .to, ...)
```

Arguments

`.x`	An object from which to dispatch.
`...`	Named clauses. The names should be types as returned by `type_of()`.
`.to`	This is useful when you switchpatch within a coercing function. If supplied, this should be a string indicating the target type. A catch-all clause is then added to signal an error stating the conversion failure. This type is prettified unless `.to` inherits from the S3 class `"AsIs"` (see `base::I()`).

See Also

switch_lang()

Examples

```
switch_type(3L,
  double = "foo",
  integer = "bar",
  "default"
)

# Use the coerce_ version to get standardised error handling when no
# type matches:
to_chr <- function(x) {
  coerce_type(x, "a chr",
    integer = as.character(x),
    double = as.character(x)
  )
}
to_chr(3L)

# Strings have their own type:
switch_type("str",
  character = "foo",
  string = "bar",
  "default"
)

# Use a fallthrough clause if you need to dispatch on all character
# vectors, including strings:
switch_type("str",
  string = ,
  character = "foo",
  "default"
)

# special and builtin functions are treated as primitive, since
# there is usually no reason to treat them differently:
switch_type(base::list,
  primitive = "foo",
  "default"
)
switch_type(base::`$`,
  primitive = "foo",
  "default"
)

# closures are not primitives:
switch_type(rlang::switch_type,
  primitive = "foo",
  "default"
)
```

sym *Create a symbol or list of symbols*

Description

These functions take strings as input and turn them into symbols. Contrarily to `as.name()`, they convert the strings to the native encoding beforehand. This is necessary because symbols remove silently the encoding mark of strings (see `set_str_encoding()`).

Usage

```
sym(x)

syms(x)
```

Arguments

`x`	A string or list of strings.

Value

A symbol for `sym()` and a list of symbols for `syms()`.

tidyeval-data *Data pronoun for tidy evaluation*

Description

This pronoun is installed by functions performing tidy evaluation. It allows you to refer to overscoped data explicitly.

Usage

```
.data
```

Format

An object of class `dictionary` of length 0.

Details

You can import this object in your package namespace to avoid `R CMD check` errors when referring to overscoped objects.

Examples

```
quo <- quo(.data$foo)
eval_tidy(quo, list(foo = "bar"))
```

type-predicates *Type predicates*

Description

These type predicates aim to make type testing in R more consistent. They are wrappers around `base::typeof()`, so operate at a level beneath S3/S4 etc.

Usage

```
is_list(x, n = NULL)

is_atomic(x, n = NULL)

is_vector(x, n = NULL)

is_integer(x, n = NULL)

is_double(x, n = NULL)

is_character(x, n = NULL, encoding = NULL)

is_logical(x, n = NULL)

is_raw(x, n = NULL)

is_bytes(x, n = NULL)

is_null(x)
```

Arguments

x	Object to be tested.
n	Expected length of a vector.
encoding	Expected encoding of a string or character vector. One of `UTF-8`, `latin1`, or `unknown`.

Details

Compared to base R functions:

- The predicates for vectors include the `n` argument for pattern-matching on the vector length.
- Unlike `is.atomic()`, `is_atomic()` does not return `TRUE` for `NULL`.
- Unlike `is.vector()`, `is_vector()` test if an object is an atomic vector or a list. `is.vector` checks for the presence of attributes (other than name).
- `is_function()` returns `TRUE` only for regular functions, not special or primitive functions.

See Also

bare-type-predicates scalar-type-predicates

`type_of` *Base type of an object*

Description

This is equivalent to `base::typeof()` with a few differences that make dispatching easier:

- The type of one-sided formulas is "quote".
- The type of character vectors of length 1 is "string".
- The type of special and builtin functions is "primitive".

Usage

```
type_of(x)
```

Arguments

`x`	An R object.

Examples

```
type_of(10L)

# Quosures are treated as a new base type but not formulas:
type_of(quo(10L))
type_of(~10L)

# Compare to base::typeof():
typeof(quo(10L))

# Strings are treated as a new base type:
type_of(letters)
type_of(letters[[1]])

# This is a bit inconsistent with the core language tenet that data
# types are vectors. However, treating strings as a different
# scalar type is quite helpful for switching on function inputs
# since so many arguments expect strings:
switch_type("foo", character = abort("vector!"), string = "result")

# Special and builtin primitives are both treated as primitives.
# That's because it is often irrelevant which type of primitive an
# input is:
typeof(list)
typeof(`$`)
type_of(list)
type_of(`$`)
```

vector-along *Create vectors matching the length of a given vector*

Description

These functions take the idea of `seq_along()` and generalise it to creating lists (`list_along`) and repeating values (`rep_along`). Except for `list_along()` and `raw_along()`, the empty vectors are filled with typed `missing` values.

Usage

```
lgl_along(.x)

int_along(.x)

dbl_along(.x)

chr_along(.x)

cpl_along(.x)

raw_along(.x)

bytes_along(.x)

list_along(.x)

rep_along(.x, .y)
```

Arguments

`.x`	A vector.
`.y`	Values to repeat.

See Also

vector-len

Examples

```
x <- 0:5
rep_along(x, 1:2)
rep_along(x, 1)
list_along(x)
```

vector-coercion *Coerce an object to a base type*

Description

These are equivalent to the base functions (e.g. `as.logical()`, `as.list()`, etc), but perform coercion rather than conversion. This means they are not generic and will not call S3 conversion methods. They only attempt to coerce the base type of their input. In addition, they have stricter implicit coercion rules and will never attempt any kind of parsing. E.g. they will not try to figure out if a character vector represents integers or booleans. Finally, they have treat attributes consistently, unlike the base R functions: all attributes except names are removed.

Usage

```
as_logical(x)

as_integer(x)

as_double(x)

as_complex(x)

as_character(x, encoding = NULL)

as_string(x, encoding = NULL)

as_list(x)
```

Arguments

`x`	An object to coerce to a base type.
`encoding`	If non-null, passed to `set_chr_encoding()` to add an encoding mark. This is only declarative, no encoding conversion is performed.

Coercion to logical and numeric atomic vectors

- To logical vectors: Integer and integerish double vectors. See `is_integerish()`.
- To integer vectors: Logical and integerish double vectors.
- To double vectors: Logical and integer vectors.
- To complex vectors: Logical, integer and double vectors.

Coercion to character vectors

`as_character()` and `as_string()` have an optional `encoding` argument to specify the encoding. R uses this information for internal handling of strings and character vectors. Note that

this is only declarative, no encoding conversion is attempted. See `as_utf8_character()` and `as_native_character()` for coercing to a character vector and attempt encoding conversion.

See also `set_chr_encoding()` and `mut_utf8_locale()` for information about encodings and locales in R, and `string()` and `chr()` for other ways of creating strings and character vectors.

Note that only `as_string()` can coerce symbols to a scalar character vector. This makes the code more explicit and adds an extra type check.

Coercion to lists

`as_list()` only coerces vector and dictionary types (environments are an example of dictionary type). Unlike `base::as.list()`, `as_list()` removes all attributes except names.

Effects of removing attributes

A technical side-effect of removing the attributes of the input is that the underlying objects has to be copied. This has no performance implications in the case of lists because this is a shallow copy: only the list structure is copied, not the contents (see `duplicate()`). However, be aware that atomic vectors containing large amounts of data will have to be copied.

In general, any attribute modification creates a copy, which is why it is better to avoid using attributes with heavy atomic vectors. Uncopyable objects like environments and symbols are an exception to this rule: in this case, attributes modification happens in place and has side-effects.

Examples

```
# Coercing atomic vectors removes attributes with both base R and rlang:
x <- structure(TRUE, class = "foo", bar = "baz")
as.logical(x)

# But coercing lists preserves attributes in base R but not rlang:
l <- structure(list(TRUE), class = "foo", bar = "baz")
as.list(l)
as_list(l)

# Implicit conversions are performed in base R but not rlang:
as.logical(l)
## Not run:
as_logical(l)

## End(Not run)

# Conversion methods are bypassed, making the result of the
# coercion more predictable:
as.list.foo <- function(x) "wrong"
as.list(l)
as_list(l)

# The input is never parsed. E.g. character vectors of numbers are
# not converted to numeric types:
as.integer("33")
## Not run:
```

```
as_integer("33")

## End(Not run)


# With base R tools there is no way to convert an environment to a
# list without either triggering method dispatch, or changing the
# original environment. as_list() makes it easy:
x <- structure(as_env(mtcars[1:2]), class = "foobar")
as.list.foobar <- function(x) abort("dont call me")
as_list(x)
```

`vector-construction` *Create vectors*

Description

The atomic vector constructors are equivalent to `c()` but allow you to be more explicit about the output type. Implicit coercions (e.g. from integer to logical) follow the rules described in vector-coercion. In addition, all constructors support splicing: if you supply bare lists or explicitly spliced lists, their contents are spliced into the output vectors (see below for details). `ll()` is a list constructor similar to `base::list()` but with splicing semantics.

Usage

```
lgl(...)

int(...)

dbl(...)

cpl(...)

chr(..., .encoding = NULL)

bytes(...)

ll(...)
```

Arguments

`...`	Components of the new vector. Bare lists and explicitly spliced lists are spliced.
`.encoding`	If non-null, passed to `set_chr_encoding()` to add an encoding mark. This is only declarative, no encoding conversion is performed.

Splicing

Splicing is an operation similar to flattening one level of nested lists, e.g. with `base::unlist(x, recursive = FALSE)` or `purrr::flatten()`. `ll()` returns its arguments as a list, just like `list()` would, but inner lists qualifying for splicing are flattened. That is, their contents are embedded in the surrounding list. Similarly, `chr()` concatenates its arguments and returns them as a single character vector, but inner lists are flattened before concatenation.

Whether an inner list qualifies for splicing is determined by the type of splicing semantics. All the atomic constructors like `chr()` have *list splicing* semantics: bare lists and explicitly spliced lists are spliced.

There are two list constructors with different splicing semantics. `ll()` only splices lists explicitly marked with `splice()`.

See Also

`ll()`

Examples

```
# These constructors are like a typed version of c():
c(TRUE, FALSE)
lgl(TRUE, FALSE)

# They follow a restricted set of coercion rules:
int(TRUE, FALSE, 20)

# Lists can be spliced:
dbl(10, list(1, 2L), TRUE)


# They splice names a bit differently than c(). The latter
# automatically composes inner and outer names:
c(a = c(A = 10), b = c(B = 20, C = 30))

# On the other hand, rlang's ctors use the inner names and issue a
# warning to inform the user that the outer names are ignored:
dbl(a = c(A = 10), b = c(B = 20, C = 30))
dbl(a = c(1, 2))

# As an exception, it is allowed to provide an outer name when the
# inner vector is an unnamed scalar atomic:
dbl(a = 1)

# Spliced lists behave the same way:
dbl(list(a = 1))
dbl(list(a = c(A = 1)))

# bytes() accepts integerish inputs
bytes(1:10)
bytes(0x01, 0xff, c(0x03, 0x05), list(10, 20, 30L))
```

```
# The list constructor has explicit splicing semantics:
ll(1, list(2))

# Note that explicitly spliced lists are always spliced:
ll(!!! list(1, 2))
```

---

vector-len *Create vectors matching a given length*

Description

These functions construct vectors of given length, with attributes specified via dots. Except for `list_len()` and `bytes_len()`, the empty vectors are filled with typed missing values. This is in contrast to the base function `base::vector()` which creates zero-filled vectors.

Usage

```
lgl_len(.n)

int_len(.n)

dbl_len(.n)

chr_len(.n)

cpl_len(.n)

raw_len(.n)

bytes_len(.n)

list_len(.n)
```

Arguments

`.n`	The vector length.

See Also

vector-along

Examples

```
list_len(10)
lgl_len(10)
```

`with_env` *Evaluate an expression within a given environment*

Description

These functions evaluate `expr` within a given environment (`env` for `with_env()`, or the child of the current environment for `locally`). They rely on `eval_bare()` which features a lighter evaluation mechanism than base R `base::eval()`, and which also has some subtle implications when evaluting stack sensitive functions (see help for `eval_bare()`).

Usage

```
with_env(env, expr)

locally(expr)
```

Arguments

`env`	An environment within which to evaluate `expr`. Can be an object with an `get_env()` method.
`expr`	An expression to evaluate.

Details

`locally()` is equivalent to the base function `base::local()` but it produces a much cleaner evaluation stack, and has stack-consistent semantics. It is thus more suited for experimenting with the R language.

Examples

```
# with_env() is handy to create formulas with a given environment:
env <- child_env("rlang")
f <- with_env(env, ~new_formula())
identical(f_env(f), env)

# Or functions with a given enclosure:
fn <- with_env(env, function() NULL)
identical(get_env(fn), env)


# Unlike eval() it doesn't create duplicates on the evaluation
# stack. You can thus use it e.g. to create non-local returns:
fn <- function() {
  g(get_env())
  "normal return"
}
g <- function(env) {
  with_env(env, return("early return"))
}
```

```
fn()


# Since env is passed to as_env(), it can be any object with an
# as_env() method. For strings, the pkg_env() is returned:
with_env("base", ~mtcars)

# This can be handy to put dictionaries in scope:
with_env(mtcars, cyl)
```

`with_handlers` *Establish handlers on the stack*

Description

Condition handlers are functions established on the evaluation stack (see `ctxt_stack()`) that are called by R when a condition is signalled (see `cnd_signal()` and `abort()` for two common signal functions). They come in two types: exiting handlers, which jump out of the signalling context and are transferred to `with_handlers()` before being executed. And inplace handlers, which are executed within the signal functions.

Usage

```
with_handlers(.expr, ...)
```

Arguments

`.expr`	An expression to execute in a context where new handlers are established. The underscored version takes a quoted expression or a quoted formula.
`...`	Named handlers. Handlers should inherit from `exiting` or `inplace`. See `exiting()` and `inplace()` for constructing such handlers. Dots are evaluated with explicit splicing.

Details

An exiting handler is taking charge of the condition. No other handler on the stack gets a chance to handle the condition. The handler is executed and `with_handlers()` returns the return value of that handler. On the other hand, in place handlers do not necessarily take charge. If they return normally, they decline to handle the condition, and R looks for other handlers established on the evaluation stack. Only by jumping to an earlier call frame can an inplace handler take charge of the condition and stop the signalling process. Sometimes, a muffling restart has been established for the purpose of jumping out of the signalling function but not out of the context where the condition was signalled, which allows execution to resume normally. See `rst_muffle()` the `muffle` argument of `inplace()` and the `mufflable` argument of `cnd_signal()`.

Exiting handlers are established first by `with_handlers()`, and in place handlers are installed in second place. The latter handlers thus take precedence over the former.

See Also

exiting(), inplace().

Examples

```
# Signal a condition with cnd_signal():
fn <- function() {
  g()
  cat("called?\n")
  "fn() return value"
}
g <- function() {
  h()
  cat("called?\n")
}
h <- function() {
  cnd_signal("foo")
  cat("called?\n")
}

# Exiting handlers jump to with_handlers() before being
# executed. Their return value is handed over:
handler <- function(c) "handler return value"
with_handlers(fn(), foo = exiting(handler))

# In place handlers are called in turn and their return value is
# ignored. Returning just means they are declining to take charge of
# the condition. However, they can produce side-effects such as
# displaying a message:
some_handler <- function(c) cat("some handler!\n")
other_handler <- function(c) cat("other handler!\n")
with_handlers(fn(), foo = inplace(some_handler), foo = inplace(other_handler))

# If an in place handler jumps to an earlier context, it takes
# charge of the condition and no other handler gets a chance to
# deal with it. The canonical way of transferring control is by
# jumping to a restart. See with_restarts() and restarting()
# documentation for more on this:
exiting_handler <- function(c) rst_jump("rst_foo")
fn2 <- function() {
  with_restarts(g(), rst_foo = function() "restart value")
}
with_handlers(fn2(), foo = inplace(exiting_handler), foo = inplace(other_handler))
```

---

with_restarts *Establish a restart point on the stack*

Description

Restart points are named functions that are established with `with_restarts()`. Once established, you can interrupt the normal execution of R code, jump to the restart, and resume execution from there. Each restart is established along with a restart function that is executed after the jump and that provides a return value from the establishing point (i.e., a return value for `with_restarts()`).

Usage

```
with_restarts(.expr, ...)
```

Arguments

`.expr`	An expression to execute with new restarts established on the stack. This argument is passed by expression and supports unquoting. It is evaluated in a context where restarts are established.
`...`	Named restart functions. The name is taken as the restart name and the function is executed after the jump. These dots are evaluated with explicit splicing.

Details

Restarts are not the only way of jumping to a previous call frame (see `return_from()` or `return_to()`). However, they have the advantage of being callable by name once established.

See Also

`return_from()` and `return_to()` for a more flexible way of performing a non-local jump to an arbitrary call frame.

Examples

```
# Restarts are not the only way to jump to a previous frame, but
# they have the advantage of being callable by name:
fn <- function() with_restarts(g(), my_restart = function() "returned")
g <- function() h()
h <- function() { rst_jump("my_restart"); "not returned" }
fn()

# Whereas a non-local return requires to manually pass the calling
# frame to the return function:
fn <- function() g(get_env())
g <- function(env) h(env)
h <- function(env) { return_from(env, "returned"); "not returned" }
fn()


# rst_maybe_jump() checks that a restart exists before trying to jump:
fn <- function() {
  g()
  cat("will this be called?\n")
}
```

```
g <- function() {
  rst_maybe_jump("my_restart")
  cat("will this be called?\n")
}

# Here no restart are on the stack:
fn()

# If a restart point called `my_restart` was established on the
# stack before calling fn(), the control flow will jump there:
rst <- function() {
  cat("restarting...\n")
  "return value"
}
with_restarts(fn(), my_restart = rst)


# Restarts are particularly useful to provide alternative default
# values when the normal output cannot be computed:

fn <- function(valid_input) {
  if (valid_input) {
    return("normal value")
  }

  # We decide to return the empty string "" as default value. An
  # altenative strategy would be to signal an error. In any case,
  # we want to provide a way for the caller to get a different
  # output. For this purpose, we provide two restart functions that
  # returns alternative defaults:
  restarts <- list(
    rst_empty_chr = function() character(0),
    rst_null = function() NULL
  )

  with_restarts(splice(restarts), .expr = {

    # Signal a typed condition to let the caller know that we are
    # about to return an empty string as default value:
    cnd_signal("default_empty_string")

    # If no jump to with_restarts, return default value:
    ""
  })
}

# Normal value for valid input:
fn(TRUE)

# Default value for bad input:
fn(FALSE)

# Change the default value if you need an empty character vector by
```

```
# defining an inplace handler that jumps to the restart. It has to
# be inplace because exiting handlers jump to the place where they
# are established before being executed, and the restart is not
# defined anymore at that point:
rst_handler <- inplace(function(c) rst_jump("rst_empty_chr"))
with_handlers(fn(FALSE), default_empty_string = rst_handler)

# You can use restarting() to create restarting handlers easily:
with_handlers(fn(FALSE), default_empty_string = restarting("rst_null"))
```

Index

!! *(quasiquotation)*, 93
!!! *(quasiquotation)*, 93
∗Topic **datasets**
    missing, 78
    tidyeval-data, 122
.Internal(), *64*
.Primitive(), *64*
.data *(tidyeval-data)*, 122
:= *(op-definition)*, 85

abort, 4
abort(), *17*, *40*, *102*, *103*, *132*
alist, *84*
are_na, 5
arg_match, 6
as.list(), *126*
as.logical(), *126*
as_bytes, 7
as_character *(vector-coercion)*, 126
as_closure *(as_function)*, 8
as_complex *(vector-coercion)*, 126
as_dictionary *(dictionary)*, 18
as_dictionary(), *38*
as_double *(vector-coercion)*, 126
as_env, 7
as_env(), *24*
as_function, 8
as_function(), *40*
as_integer *(vector-coercion)*, 126
as_list *(vector-coercion)*, 126
as_logical *(vector-coercion)*, 126
as_native_character
        (as_utf8_character), 12
as_native_character(), *127*
as_native_string *(as_utf8_character)*, 12
as_overscope, 9
as_overscope(), *24*, *39*
as_pairlist, 11
as_quosure, 11
as_quosureish *(as_quosure)*, 11
as_string *(vector-coercion)*, 126
as_utf8_character, 12
as_utf8_character(), *127*
as_utf8_string *(as_utf8_character)*, 12
as_utf8_string(), *111*

bare, *21*, *128*, *129*
bare-type-predicates, 13, *106*, *124*
base environment, *10*
base::.Internal(), *64*
base::append(), *92*
base::as.list(), *127*
base::assign(), *26*
base::attach(), *107*
base::bquote(), *41*
base::c(), *92*
base::delayedAssign(), *26*
base::do.call(), *55*
base::eval(), *36*, *37*, *115*, *131*
base::formals(), *60*
base::I(), *119*, *120*
base::ifelse(), *87*
base::is.integer(), *65*
base::is.na(), *5*
base::length(), *54*
base::library(), *107*
base::list(), *128*
base::local(), *131*
base::makeActiveBinding(), *26*
base::match.arg(), *6*
base::match.call(), *15*, *77*
base::message(), *4*
base::missing(), *79*
base::pairlist(), *89*
base::parent.frame(), *36*
base::parse(), *91*
base::quote(), *11*, *84*
base::return(), *36*, *101*
base::search(), *107*
base::stop(), *4*, *16*, *40*

base::substitute(), *41*
base::tryCatch(), *40*
base::typeof(), *123, 124*
base::vector(), *130*
base::warning(), *4*
base_env *(*scoped_env*)*, 107
base_env(), *59*
bound, *71*
bytes *(*vector-construction*)*, 128
bytes_along *(*vector-along*)*, 125
bytes_len *(*vector-len*)*, 130

c(), *128*
call node, *56*
call_depth *(*stack*)*, 114
call_frame *(*stack*)*, 114
call_frame(), *15*
call_inspect, 15
call_stack *(*stack*)*, 114
callable, *71, 72*
caller frame, *6*
caller_env, 15
caller_fn *(*caller_env*)*, 15
caller_frame *(*caller_env*)*, 15
capture, *72*
child_env *(*env*)*, 24
chr *(*vector-construction*)*, 128
chr(), *113*, *127*
chr_along *(*vector-along*)*, 125
chr_encoding *(*set_chr_encoding*)*, 110
chr_len *(*vector-len*)*, 130
closure, *8*, *27*, *29–35*, *52*
closures, *26*
cnd_abort *(*cnd_signal*)*, 16
cnd_abort(), *5*, *103*
cnd_error *(*new_cnd*)*, 82
cnd_message *(*new_cnd*)*, 82
cnd_signal, 16
cnd_signal(), *82*, *105*, *132*
cnd_warning *(*new_cnd*)*, 82
coerce_class *(*switch_type*)*, 120
coerce_lang *(*switch_lang*)*, 119
coerce_type *(*switch_type*)*, 120
constructed calls, *48*
cpl *(*vector-construction*)*, 128
cpl_along *(*vector-along*)*, 125
cpl_len *(*vector-len*)*, 130
ctxt_depth *(*stack*)*, 114
ctxt_frame *(*stack*)*, 114
ctxt_stack *(*stack*)*, 114
ctxt_stack(), *36*, *40*, *47*, *52*, *63*, *108*, *117*, *132*
current_frame *(*stack*)*, 114

dbl *(*vector-construction*)*, 128
dbl_along *(*vector-along*)*, 125
dbl_len *(*vector-len*)*, 130
definition, *11*, *61*
definitions, *61*
dictionaries, *24*
dictionary, 18
dots_definitions, 19
dots_list, 20
dots_list(), *22*, *113*
dots_n, 22
dots_splice *(*dots_list*)*, 20
dots_splice(), *22*, *113*
dots_values, 22
duplicate, 23
duplicate(), *89*, *127*

empty environment, *24*, *30*, *95*
empty_env, 23
empty_env(), *7*, *34*
enexpr *(*expr*)*, 41
enquo *(*quosure*)*, 95
enquo(), *40*, *95*
env, 24
env(), *27*, *31*, *34*, *47*
env_bind, 26
env_bind(), *25*, *29*, *35*
env_bind_exprs *(*env_bind*)*, 26
env_bind_fns *(*env_bind*)*, 26
env_bury, 29
env_bury(), *55*
env_clone, 30
env_depth, 30
env_get, 31
env_has, 32
env_has(), *25*
env_inherits, 32
env_names, 33
env_parent, 34
env_parents *(*env_parent*)*, 34
env_tail *(*env_parent*)*, 34
env_unbind, 35
env_unbind(), *29*
environment, *95*

eval_bare, 36
eval_bare(), *10*, *37*, *55*, *131*
eval_tidy, 37
eval_tidy(), *10*, *19*, *39*, *96*
eval_tidy_, 39
eval_tidy_(), *10*
exiting, 40
exiting(), *16*, *101*, *132, 133*
explicit splicing, *16*, *26*, *72*, *75*, *80*, *82*, *100*, *104*, *132*, *134*
explicit splicing semantics, *24*, *109*
explicitly spliced, *128, 129*
expr, 41
expr(), *93*, *97*
expr_interp, 43
expr_label, 44
expr_label(), *51*, *99*
expr_name *(*expr_label*)*, 44
expr_text *(*expr_label*)*, 44
expr_text(), *19*, *42*, *51*
expression, *11*
expression(), *59*
expressions, *71*, *95*
exprs *(*expr*)*, 41
exprs(), *21*, *26*
exprs_auto_name, 42
exprs_auto_name(), *19*

f_env *(*f_rhs*)*, 50
f_env<- *(*f_rhs*)*, 50
f_label *(*f_text*)*, 51
f_label(), *99*
f_lhs *(*f_rhs*)*, 50
f_lhs<- *(*f_rhs*)*, 50
f_name *(*f_text*)*, 51
f_rhs, 50
f_rhs<- *(*f_rhs*)*, 50
f_text, 51
f_text(), *99*
flatten, 45
flatten(), *114*
flatten_chr *(*flatten*)*, 45
flatten_cpl *(*flatten*)*, 45
flatten_dbl *(*flatten*)*, 45
flatten_if *(*flatten*)*, 45
flatten_if(), *22*
flatten_int *(*flatten*)*, 45
flatten_lgl *(*flatten*)*, 45
flatten_raw *(*flatten*)*, 45

fn_env, 47
fn_env<- *(*fn_env*)*, 47
fn_fmls, 48
fn_fmls(), *60*, *63, 64*, *73*
fn_fmls_names *(*fn_fmls*)*, 48
fn_fmls_names(), *73*
fn_fmls_syms *(*fn_fmls*)*, 48
formals(), *63*
formula, *11*, *61*
frame_position, 49
friendly_type, 50

get_env, 52
get_env(), *27*, *29*, *49*, *102*, *131*
get_expr *(*set_expr*)*, 111
global_env *(*scoped_env*)*, 107
global_frame *(*stack*)*, 114
global_frame(), *108*

has_length, 53
has_name, 54
have_name *(*is_named*)*, 67

implicit splicing, *26*
inform *(*abort*)*, 4
inform(), *17*
inplace *(*exiting*)*, 40
inplace(), *5*, *16, 17*, *100, 101*, *105*, *132, 133*
int *(*vector-construction*)*, 128
int_along *(*vector-along*)*, 125
int_len *(*vector-len*)*, 130
invoke, 55
is_atomic *(*type-predicates*)*, 123
is_atomic(), *14*
is_bare_atomic *(*bare-type-predicates*)*, 13
is_bare_bytes *(*bare-type-predicates*)*, 13
is_bare_character *(*bare-type-predicates*)*, 13
is_bare_double *(*bare-type-predicates*)*, 13
is_bare_env *(*is_env*)*, 58
is_bare_formula *(*is_formula*)*, 61
is_bare_integer *(*bare-type-predicates*)*, 13
is_bare_integerish *(*is_integerish*)*, 65
is_bare_list *(*bare-type-predicates*)*, 13
is_bare_logical *(*bare-type-predicates*)*, 13

is_bare_numeric (bare-type-predicates), 13
is_bare_numeric(), *65*
is_bare_raw (bare-type-predicates), 13
is_bare_string (bare-type-predicates), 13
is_bare_vector (bare-type-predicates), 13
is_binary_lang (is_lang), 66
is_bytes (type-predicates), 123
is_call_stack (is_stack), 70
is_callable, 56
is_callable(), *72*
is_character (type-predicates), 123
is_chr_na (are_na), 5
is_closure (is_function), 63
is_condition, 57
is_copyable, 57
is_cpl_na (are_na), 5
is_dbl_na (are_na), 5
is_definition (op-definition), 85
is_dictionary (dictionary), 18
is_dictionaryish (is_named), 67
is_dictionaryish(), *7*
is_double (type-predicates), 123
is_empty, 58
is_env, 58
is_eval_stack (is_stack), 70
is_expr, 59
is_expr(), *33*, *41*, *42*, *66*, *119*
is_false (is_true), 71
is_formula, 61
is_formula(), *69*
is_formulaish (is_formula), 61
is_formulaish(), *69*
is_frame, 62
is_function, 63
is_function(), *8*, *48*
is_installed, 64
is_int_na (are_na), 5
is_integer (type-predicates), 123
is_integerish, 65
is_integerish(), *126*
is_lang, 66
is_lang(), *60*, *69*
is_lgl_na (are_na), 5
is_list (type-predicates), 123
is_logical (type-predicates), 123
is_missing (missing_arg), 79
is_na (are_na), 5
is_named, 67
is_node (is_pairlist), 68
is_null (type-predicates), 123
is_null(), *5*
is_pairlist, 68
is_primitive (is_function), 63
is_primitive_eager (is_function), 63
is_primitive_lazy (is_function), 63
is_quosure, 69
is_quosure(), *11*, *12*, *61*
is_quosureish (is_quosure), 69
is_quosureish(), *61*
is_quosures (dots_definitions), 19
is_raw (type-predicates), 123
is_scalar_atomic (scalar-type-predicates), 106
is_scalar_bytes (scalar-type-predicates), 106
is_scalar_character (scalar-type-predicates), 106
is_scalar_double (scalar-type-predicates), 106
is_scalar_integer (scalar-type-predicates), 106
is_scalar_integerish (is_integerish), 65
is_scalar_list (scalar-type-predicates), 106
is_scalar_logical (scalar-type-predicates), 106
is_scalar_raw (scalar-type-predicates), 106
is_scalar_vector (scalar-type-predicates), 106
is_scoped (scoped_env), 107
is_spliced (splice), 113
is_spliced_bare (splice), 113
is_stack, 70
is_string (scalar-type-predicates), 106
is_symbol, 70
is_symbolic (is_expr), 59
is_symbolic(), *11*
is_syntactic_literal (is_expr), 59
is_true, 71
is_unary_lang (is_lang), 66
is_vector (type-predicates), 123

lang, 71

lang(), *74*, *75*
lang_args, 73
lang_args(), *48*, *74*, *75*
lang_args_names *(*lang_args*)*, 73
lang_args_names(), *48*
lang_fn, 74
lang_fn(), *77*
lang_head, 74
lang_head(), *72*, *89*
lang_modify, 75
lang_name, 76
lang_name(), *74*
lang_standardise, 77
lang_tail *(*lang_head*)*, 74
lang_tail(), *60*, *72*, *73*, *89*
lang_type_of *(*switch_lang*)*, 119
lang_type_of(), *50*, *72*
language object, *66*
lexical enclosure, *11*
lgl *(*vector-construction*)*, 128
lgl_along *(*vector-along*)*, 125
lgl_len *(*vector-len*)*, 130
list_along *(*vector-along*)*, 125
list_len *(*vector-len*)*, 130
ll *(*vector-construction*)*, 128
ll(), *20*, *113*, *129*
locally *(*with_env*)*, 131

maybe_missing *(*missing_arg*)*, 79
missing, 78, *130*
missing argument, *95*
missing types, *5*
missing_arg, 79
mode, *72*
modify, 80
mut_attrs *(*set_attrs*)*, 109
mut_latin1_locale *(*mut_utf8_locale*)*, 81
mut_mbcs_locale *(*mut_utf8_locale*)*, 81
mut_node_caar *(*pairlist*)*, 88
mut_node_cadr *(*pairlist*)*, 88
mut_node_car *(*pairlist*)*, 88
mut_node_car(), *23*
mut_node_cdar *(*pairlist*)*, 88
mut_node_cddr *(*pairlist*)*, 88
mut_node_cdr *(*pairlist*)*, 88
mut_node_cdr(), *23*
mut_node_tag *(*pairlist*)*, 88
mut_utf8_locale, 81
mut_utf8_locale(), *12*, *110*, *111*, *127*

na_chr *(*missing*)*, 78
na_cpl *(*missing*)*, 78
na_dbl *(*missing*)*, 78
na_int *(*missing*)*, 78
na_lgl *(*missing*)*, 78
names2, 81
new_cnd, 82
new_cnd(), *16*
new_definition *(*op-definition*)*, 85
new_environment *(*env*)*, 24
new_formula, 83
new_function, 84
new_language *(*lang*)*, 71
new_overscope *(*as_overscope*)*, 9
new_quosure *(*quosure*)*, 95
new_quosure(), *83*
node *(*pairlist*)*, 88
node_caar *(*pairlist*)*, 88
node_cadr *(*pairlist*)*, 88
node_car *(*pairlist*)*, 88
node_car(), *74*
node_cdar *(*pairlist*)*, 88
node_cddr *(*pairlist*)*, 88
node_cdr *(*pairlist*)*, 88
node_cdr(), *74*
node_tag *(*pairlist*)*, 88
ns_env, 85
ns_env_name *(*ns_env*)*, 85
ns_imports_env *(*ns_env*)*, 85

op-definition, 85
op-get-attr, 86
op-na-default, 87
op-null-default, *87*, 87
overscope, *37*
overscope_clean *(*as_overscope*)*, 9
overscope_eval_next *(*as_overscope*)*, 9
overscope_eval_next(), *40*

pairlist, *11*, *71*, *72*, *75*, 88, *119*
pairlists, *72*
parent.frame(), *114*
parse_expr, 90
parse_expr(), *59*
parse_exprs *(*parse_expr*)*, 90
parse_quosure *(*parse_expr*)*, 90
parse_quosures *(*parse_expr*)*, 90
parses, *72*
pkg_env *(*scoped_env*)*, 107

pkg_env(), *7*, *85*
pkg_env_name *(*scoped_env*)*, 107
prepend, 92
prim_name, 92

quasiquotation, *38*, *41*, 93, *96, 97*, *114*
quo *(*quosure*)*, 95
quo(), *11*, *19*, *38*, *41–43*, *93*, *114*
quo-predicates, 94
quo_expr, 99
quo_is_lang *(*quo-predicates*)*, 94
quo_is_missing *(*quo-predicates*)*, 94
quo_is_null *(*quo-predicates*)*, 94
quo_is_symbol *(*quo-predicates*)*, 94
quo_is_symbolic *(*quo-predicates*)*, 94
quo_label *(*quo_expr*)*, 99
quo_name *(*quo_expr*)*, 99
quo_text *(*quo_expr*)*, 99
quo_text(), *42*
quos *(*dots_definitions*)*, 19
quos(), *41*, *95, 96*
quos_auto_name *(*exprs_auto_name*)*, 42
quosure, *9*, *27*, *29–35*, *52*, *69*, 95
quosureish, *93*
quosures, *8*
quosures *(*dots_definitions*)*, 19

raw_along *(*vector-along*)*, 125
raw_len *(*vector-len*)*, 130
rep_along *(*vector-along*)*, 125
restarting, 100
restarting(), *41*
return_from, 101
return_from(), *40*, *134*
return_to *(*return_from*)*, 101
return_to(), *134*
rst_abort, 102
rst_abort(), *5*, *16*, *40*
rst_exists *(*rst_list*)*, 103
rst_jump *(*rst_list*)*, 103
rst_jump(), *16*, *40*, *103*
rst_list, 103
rst_maybe_jump *(*rst_list*)*, 103
rst_muffle, 104
rst_muffle(), *5*, *17*, *104*, *132*

scalar-type-predicates, *14*, 106, *124*
scoped_env, 107
scoped_envs *(*scoped_env*)*, 107
scoped_names *(*scoped_env*)*, 107
scoped_names(), *23*
seq2, 108
seq2_along *(*seq2*)*, 108
seq_along(), *125*
set_attrs, 109
set_chr_encoding, 110
set_chr_encoding(), *33*, *118*, *126–128*
set_env *(*get_env*)*, 52
set_expr, 111
set_names, 112
set_str_encoding *(*set_chr_encoding*)*, 110
set_str_encoding(), *81*, *122*
shallow copy, *109*
splice, 113
splice(), *21, 22*, *129*
splicing, *21*
squash *(*flatten*)*, 45
squash_chr *(*flatten*)*, 45
squash_cpl *(*flatten*)*, 45
squash_dbl *(*flatten*)*, 45
squash_if *(*flatten*)*, 45
squash_int *(*flatten*)*, 45
squash_lgl *(*flatten*)*, 45
squash_raw *(*flatten*)*, 45
stack, 114
stack_trim, 117
stack_trim(), *115*
stats::setNames(), *112*
str_encoding *(*set_chr_encoding*)*, 110
string, 118
string(), *127*
structure(), *109*
switch, *120*
switch_class *(*switch_type*)*, 120
switch_lang, 119
switch_lang(), *121*
switch_type, 120
sym, 122
sym(), *66*
symbolic, *71*
symbolic objects, *56*
syms *(*sym*)*, 122

tidy evaluation, *122*
tidyeval-data, 122
type, *72*
type-predicates, *14*, *106*, 123
type_of, *120*, 124

type_of(), *50*, *120*

uncopyable, *25*, *109*
unlist(), *45*
unquoting, *134*
UQ *(*quasiquotation*)*, 93
UQE *(*quasiquotation*)*, 93
UQS *(*quasiquotation*)*, 93

vector-along, *78*, 125
vector-coercion, 126, *128*
vector-construction, *114*, 128
vector-len, 130

warn *(*abort*)*, 4
warn(), *17*
with_env, 131
with_handlers, 132
with_handlers(), *5*, *17*, *40*, *41*, *82*, *102*
with_restarts, 133
with_restarts(), *17*, *100*, *104*