Package ‘elixir’

December 2, 2025

Type Package

Title Transmutation of Languages

Version 0.1.1

Description Tools for transforming 'R' expressions. Provides functions for finding, extracting, and replacing patterns in 'R' language objects, similarly to how regular expressions can be used to find, extract, and replace patterns in text. Also provides functions for generating code using specially-formatted template files and for translating 'R' expressions into similar expressions in other programming languages. The package may be helpful for advanced uses of 'R' expressions, such as developing domain-specific languages.

URL https://github.com/nicholasdavies/elixir,
https://nicholasdavies.github.io/elixir/

BugReports https://github.com/nicholasdavies/elixir/issues

License MIT + file LICENSE

Encoding UTF-8

Imports rlang, stringr, glue

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

RoxygenNote 7.3.2

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Author Nicholas Davies [cre, aut, cph] (ORCID: <https://orcid.org/0000-0002-1740-1412>)

Maintainer Nicholas Davies <nicholas.davies@lshtm.ac.uk>

Repository CRAN

Date/Publication 2025-12-02 13:00:02 UTC

Contents

`elixir` `elixir`: *Transmutation of languages*

Description

`elixir` is a set of tools for transforming R expressions, including into other programming languages.

Details

One of the neat features of R is that you can use the language to inspect itself. Expressions, functions, indeed entire R scripts can be examined and manipulated just like any list, data.frame, or other R object.

However, the syntax for manipulating R language objects is a little tricky. Packages such as `rlang` help to make this task easier. `elixir` makes a few extra shortcuts available, and is geared for advanced R users.

`elixir` provides functions for finding, extracting, and replacing patterns in 'R' language objects, similarly to how regular expressions can be used to find, extract, and replace patterns in text. It also provides functions for generating code using specially-formatted template files and for translating 'R' expressions into similar expressions in other programming languages.

The package may be helpful for advanced uses of 'R' expressions, such as developing domain-specific languages.

Find and replace for language objects

Sometimes you want to detect certain patterns within an expression or list of expressions, or easily replace a certain pattern with another. When working with strings, regular expressions are a handy way of accomplishing such tasks. `elixir` provides a sort of "regular expressions for R expressions" functionality through the functions `expr_match()`, `expr_replace()`, and the "shortcut" functions `expr_count()`, `expr_detect()`, `expr_extract()`, and `expr_locate()`.

Other elixir features

The function expr_apply() allows you to transform and extract information from nested list structures which contain expressions, so if you have a big structure and you want to check all the variable names or make certain replacements, this may be useful.

expr_sub() offers an interface for extracting or replacing part of an expression; the one advantage this has over [[is that it allows you to use NULL as the index, which gives back the whole expression.

lang2str() does the opposite of base::str2lang(); it is like deparse1() which is new since R 4.0.0, but with collapse = "" instead of collapse = " ".

Finally, meld(), translate(), and reindent() are various experimental functions for constructing code using R.

---

elixir-expression — *Expressions in* elixir

Description

elixir is primarily a package for working with what it calls "expressions", in the sense of any R object for which rlang::is_expression() returns TRUE. This includes calls, like the results of evaluating quote(f(x)) or quote(a:b), symbols like quote(z), and syntactic literals like 2.5, "hello", NULL, FALSE, and so on. In many cases, you can also use elixir to work with formulas, even though rlang::is_expression() returns FALSE for formulas.

This is not to be confused with the built-in type base::expression, which is essentially a special way of storing a vector of multiple "expressions". elixir does not use this type; see expr_list() instead.

Usage

```
expr_list(number = { `.A:numeric` } ? { `.A:integer` },
    string = { `.A:character` }, symbol = { `.A:name` })
expr_match({ 1 * 2 }, ~{ .A * .B })
expr_match({ 1 * 2 }, { `.A:numeric` })
expr_replace({ y = a*x^3 + b*x^2 + c*x^1 + d*x^0 },
    { ..X ^ ..N }, { pow(..X, ..N) })
```

Specifying expressions in elixir

The elixir package functions starting with expr_ work with expressions. These functions all accept a special (optional) syntax for specifying expressions which involves the symbols {}, ?, and ~, as well as the rlang injection operator, !! and splice operator, !!!).

With base R, if you want to store an expression such as x + y in a variable or pass it to a function, you need to use base::quote() or rlang::expr(), but any Elixir expr_ function will also accept an "expression literal" wrapped in braces, {}.

So, for example, rather than

```
translate(quote(x ^ y), "C++")
```

you can write

`translate({ x ^ y }, "C++")`.

This only works if the braces are provided "directly"; that is, in
`expr <- quote({ x ^ y }); translate(expr, "C++")`,

the braces are not interpreted in any special way.

Anything between the braces essentially gets put through `rlang::expr()`, so you can use `!!` (i.e. rlang::injection-operator) and `!!!` (i.e. rlang::splice-operator). There is an `env` parameter to all relevant `elixir` functions, defaulting to `parent.frame()`, in which these injection operations are evaluated.

Special syntax for patterns and replacements

Additionally, some functions (`expr_match()`, `expr_count()`, `expr_detect()`, `expr_extract()`, `expr_locate()`, and `expr_replace()`) take `pattern` and/or `replacement` arguments to specify patterns to match to an expression and/or replacement expressions to replace those matches with.

For both `pattern` and `replacement` arguments, you can use the question mark operator ? to specify *alternatives*. For example, to match *either* the token `cat` or dog, you can use

`expr_match(expr, { cat } ? { dog })`.

You can chain together as many alternatives as are needed. Alternatively, if you have a list of expressions `z`, you can use a single question mark before the name of the list, like so:

`expr_match(expr, ?z)`

and `elixir` will treat the list as a set of alternatives. When using `expr_replace()` with a set of alternatives as the pattern, the replacement needs to be either a single expression, or a set of alternative expressions which has the same number of alternatives as in the pattern.

You can also use the tilde operator ~ to specify that a given pattern should be "anchored" at the top level of an expression, and will not "recurse into" the expression. For example, in

```
exprs = expr_list(2, 5, {1 + 4})
expr_match(exprs, ~{ `.A:numeric` })
```

only the numbers 2 and 5 will match. However, in

```
exprs = expr_list(2, 5, {1 + 4})
expr_match(exprs, { `.A:numeric` })
```

all numbers 2, 5, 1 and 4 will match, because the `pattern` can recurse into the third expression `1 + 4`.

elixir-rules — *Rules for understanding languages*

Description

Several elixir functions – namely meld(), reindent(), and translate() – take an argument rules which assists those functions in interpreting their arguments.

Details

In all cases, rules can either be a character string identifying a set of built-in rules for a specific language or purpose – currently, elixir accepts "C", "C++", "Lua", or "R" – or a list with elements required for interpretation.

elixir:::ruleset contains the built-in rules. Passing an empty list() as the rules argument to an elixir function will cause it to complain about the missing components, which is one way of discerning what is needed for a given function, but usually these error messages do not quite cover all details of what is needed.

expr_apply — *Apply a function over expressions*

Description

Recursively apply a function over an expression, or any expression elements of a list, and optionally the subexpressions within any expressions.

Usage

```
expr_apply(
  x,
  f,
  depth = Inf,
  into = FALSE,
  order = c("pre", "post"),
  how = c("replace", "unlist", "unique"),
  env = parent.frame()
)
```

Arguments

x	The R object; can be an expression, a formula, or a list of arbitrary nestedness potentially containing multiple expressions or formulas.
f	Function to apply to all expressions within x; takes 1 to 3 arguments.
depth	How many levels to recurse into lists; default is Inf.

into
: Whether to recurse into expressions. Can be `TRUE` to visit all subexpressions, `FALSE` to not recurse, or `"leaves"` to recurse and only apply `f` to terminal nodes of expressions (i.e. the symbols and syntactic literals comprising the expressions).

order
: Whether a parent node is visited before ("pre") or after ("post") its children (the terminology comes from pre-order and post-order depth-first search). This only has an effect if `into == TRUE`.

how
: How to structure the result.

env
: Environment for injections in `x` (see expression).

Details

The function `f` can take one to three arguments. The first argument is the expression itself for `f` to apply to, and `f` should return some kind of replacement for, or modified version of, this argument.

The second argument is a list with information about the name of the expression in the list `x` and of its parents. Specifically, the first element of the list is the name of the expression, the second element of the list is the name of the "parent" of the expression, and so on. If any elements in this chain are unnamed, an integer is provided as the name. If the expression is within another expression (which only happens with `into = TRUE`), this is signalled as a `NULL` at the top of the list, one for each level of recursion into the expression.

The third argument is an integer vector, the index into `x` where `f` is currently operating. This is suitable for use with `expr_sub()`.

Value

If `how = "replace"` (the default), the original object `x` with `f` applied to expressions within it. If `how = "unlist"`, the same but with `unlist()` applied to it. If `how = "unique"`, first `unlist()` then `unique()` are applied.

Examples

```
expr_apply(list(quote(a + b), quote(c)), function(x) all.vars(x), how = "unlist")
```

---

expr_list *Make a list of expressions*

Description

Constructs a list of expressions, with support for `elixir`'s special expression syntax (expression literals with `{}` or `~{}`, and alternatives with `?`).

Usage

```
expr_list(..., env = parent.frame())

## S3 method for class 'expr_list'
xl[i]

## S3 replacement method for class 'expr_list'
xl[i] <- value
```

Arguments

...	Expressions to include in the list. If the arguments are named, these will be passed on to the returned list.
env	Environment for injections in ... (see expression).
xl	An expr_list.
i	Index for subsetting the expr_list; an integer, numeric, logical, or character vector (for named expr_lists) interpreted in the usual R way.
value	Replacement; an expr_list, an expression, or a list of expressions.

Details

Be aware that using the [[indexing operator on an object of class expr_list discards information about whether that element of the list is marked as anchored. In other words, if xl <- expr_list({.A}, ~{.A}), then xl[[1]] and xl[[2]] are both equal to the "bare" symbol .A, so the information that the second element of the list is anchored has been lost. Consequently, in e.g. expr_match(expr, xl[[2]]), it will be as though the tilde isn't there, and xl[[2]] will not just match with the top level of expr as was probably intended. Use the [operator instead, which retains anchoring information; expr_match(expr, xl[2]) will work as expected.

Note that when you replace part of an expr_list with another expr_list, the anchoring information from the "replacement" expr_list is copied over, while replacing part of an expr_list with an expression or a "plain" list of expressions retains the existing anchoring information.

Value

A list of expressions, of class expr_list.

Examples

```
expr_list(
   ~{ 1 + 1 = 2 } ? ~{ 2 + 2 = 4 },
   ~{ y = a * x + b },
   { .A }
)

# There is support for rlang's injection operators.
var = as.name("myvar")
expr_list({ 1 }, { !!var }, { (!!var)^2 })
```

expr_match — *Find patterns in expressions*

Description

Match and extract patterns in an expression or a list of expressions.

Usage

```
expr_match(expr, pattern, n = Inf,
    dotnames = FALSE, env = parent.frame())

expr_count(expr, pattern, n = Inf, env = parent.frame())
expr_detect(expr, pattern, n = Inf, env = parent.frame())
expr_extract(expr, pattern, what = "match", n = Inf,
    dotnames = FALSE, gather = FALSE, env = parent.frame())
expr_locate(expr, pattern, n = Inf, gather = FALSE,
    env = parent.frame())
```

Arguments

`expr`	Input. An expression, expr_list, or `list()` of expressions. Also works with formulas or lists of formulas.
`pattern`	Pattern to look for. An expression, a length-one expr_list, or a length-one list of expressions. The question mark syntax (see expression) can be used to specify alternatives.
`n`	Maximum number of matches to make in each expression; default is `Inf`.
`dotnames`	Normally, patterns like `.A`, `..B`, `...C`, etc, are named just `A`, `B`, `C`, etc., in the returned matches, without the dot(s) before each name. With `dotnames = TRUE`, the dots are kept.
`env`	Environment for injections in `expr`, `pattern` (see expression).
`what`	(`expr_extract` only) Name of the pattern to extract (or `"match"`, the default, to extract the entire match).
`gather`	(`expr_extract` and `expr_locate` only) Whether to only return the successful matches, in a single unnested list.

Value

`expr_match` returns, for each expression in `expr`, either `NULL` if there is no match, or an object of class `expr_match` if there is a match. If `expr` is a single expression, just a single `NULL` or `expr_match` object will be returned, but if `expr` is a list of expressions, then a list of all results will be returned.

An `expr_match` object is a list containing the elements `alt` (if the pattern contains several alternatives), `match`, `loc`, and further elements corresponding to the capture tokens in `pattern` (see below).

For return values of `expr_count`, `expr_detect`, `expr_extract`, and `expr_locate`, see below.

Details

All of these functions are used to check whether an expression matches a specific pattern, and if it does, retrieve the details of the match. These functions are inspired by similar functions in the `stringr` package.

Details for expr_match

`expr_match` is the most general of the bunch. As an example, suppose you had an expression containing the sum of two numbers (e.g. `3.14159 + 2.71828`) and you wanted to extract the two numbers. You could use the pattern `{ .A + .B }` to extract the match:

```
expr_match({ 3.14159 + 2.71828 }, { .A + .B })
```

This gives you a list containing all the matches found. In this case, there is one match, the details of which are contained in an object of class `expr_match`. This object contains the following elements:

- `match = quote(3.14159 + 2.71828)`, the entire match;
- `loc = NULL`, the location of the match within the expression;
- `A = 3.14159`, the part of the match corresponding to the *capture token* `.A`;
- `B = 2.71828`, the part of the match corresponding to the *capture token* `.B`.

We can also use a list of expressions for `expr`, as in:

```
ex <- expr_list({ x + y }, { kappa + lambda }, { p * z })
expr_match(ex, { .A + .B })
```

This returns a list with one entry for each element of the list `ex`; for the expressions that match (`ex[[1]]` and `ex[[2]]`) an `expr_match` object is returned, while for the expression that does not match (`ex[[3]]`), `NULL` is returned.

Pattern syntax

The `pattern` expression (e.g. `{.A + .B}` in the above) follows a special syntax.

Capture tokens:

First, these patterns can contain *capture tokens*, which are names starting with one to three periods and match to the following:

- `.A` matches any single token
- `..A` matches any sub-expression
- `...A` matches any number of function arguments

Above, "A" can be any name consisting of an alphabetical character (`a-z`, `A-Z`) followed by any number of alphanumeric characters (`a-z`, `A-Z`, `0-9`), underscores (`_`), or dots (`.`). This is the name given to the match in the returned list. Alternatively, it can be any name starting with an underscore (e.g. so the entire token could be `._` or `..._1`), in which case the match is made but the capture is discarded.

Additionally, the single-token pattern (e.g. `.A`) can be extended as follows:

- Use `` `.A:classname` `` to require that the class of the object be "classname" (or contain "classname" if the object has multiple classes); so e.g. `` `.A:name` `` matches a single name (i.e. symbol).
- Use `` `.A/regexp` `` to require a regular expression match `regexp`; so e.g. `` `.A:name/ee` `` will match symbols with two consecutive lowercase letter 'e's;
- Use `` `.A|test` `` to require that the expression `test` evaluates to `TRUE`, where `.` can be used as a stand-in for the matched token; so e.g. `` `.A:numeric|.>5` `` will match numbers greater than 5.

The `regexp` and `test` specifiers cannot be used together, and have to come after the `classname` specifier if one appears. These special syntaxes require the whole symbol to be wrapped in backticks, as in the examples above, so that they parse as symbols.

Matching function arguments:

If you wish to match a single, unnamed function argument, you can use a capture token of the form `.A` (single-token argument) or `..B` (expression argument). To match all arguments, including named ones, use a capture token of the form `...C`. For example, these all match:

```
expr_match({ myfunc() }, { .F() })
expr_match({ myfunc(1) }, { .F(.X) })
expr_match({ myfunc(1 + 1) }, { myfunc(..X) })
expr_match({ myfunc(1, 2) }, { .F(.X, .Y) })
expr_match({ myfunc() }, { myfunc(...A) })
expr_match({ myfunc(1) }, { .F(...A) })
expr_match({ myfunc(2, c = 3) }, { myfunc(...A) })
```

but these do not:

```
expr_match({ myfunc() }, { .F(.X) })
expr_match({ myfunc() }, { .F(..X) })
expr_match({ myfunc(a = 1) }, { .F(.X) })
expr_match({ myfunc(a = 1 + 1) }, { .F(..X) })
expr_match({ myfunc(1,2) }, { .F(..X) })
expr_match({ myfunc(a = 1, b = 2) }, { .F(...X, ...Y) })
```

There may be support for named arguments in patterns in the future, e.g. a pattern such as `{ f(a = .X) }` that would match an expression like `{ f(a = 1) }`, but that is currently not supported. So currently you can only match named function arguments using the `...X` syntax.

Anchoring versus recursing into expressions:

If you want your anchor your pattern, i.e. ensure that the pattern will only match at the "outer level" of your expression(s), without matching to any sub-expressions within, use a tilde (~) outside the braces (see expression for details). For example, `expr_match({1 + 2 + 3 + 4}, ~{..A + .B})` only gives one match, to the addition at the outermost level of `1 + 2 + 3` plus `4`, but `expr_match({1 + 2 + 3 + 4}, {..A + .B})` also matches to the inner additions of `1 + 2` plus `3` and `1` plus `2`.

Alternatives:

Finally, `pattern` can be a series of alternatives, using the operator `?` for specifying alternatives (see expression for details). Results from the first matching pattern among these alternatives will be returned, and the returned `expr_match` object will include a special element named "alt" giving the index of the matching alternative (see examples).

Details for expr_count, expr_detect, expr_extract, and expr_locate

These shortcut functions return only some of the information given by expr_match, but often in a more convenient format.

expr_count returns an integer vector with one element for every expression in expr, each element giving the number of matches of pattern found.

expr_detect returns a logical vector with one element for every expression in expr, each element giving whether at least one match of pattern was found.

expr_extract returns, for each expression in expr, a list of all the complete matches. Or, by specifing a capture token name in the argument which, those can be extracted instead. For example:

```
expr_extract(expr_list({(a+b)+(x+y)},
    {"H"*"I"}, {3+4}), {.A + .B}, "A")
```

gives list(list(quote(a), quote(x)), NULL, list(3)).

Using gather = TRUE with expr_extract returns only the succesful matches in a single, unnested list; so the above call to expr_extract with gather = TRUE would give list(quote(a), quote(x), 3).

Finally, expr_locate is similar to expr_extract but it returns the location within expr of each successful match.

See Also

expr_replace() to replace patterns in expressions.

Examples

```
expr_match({ 1 + 2 }, { .A + .B })

# match to one of several alternatives
expr_match({ 5 - 1 }, { .A + .B } ? { .A - .B })
```

---

expr_replace — *Replace patterns within expressions*

Description

Match and replace elements of patterns in an expression or a list of expressions.

Usage

```
expr_replace(expr, ..., patterns, replacements,
    n = Inf, env = parent.frame())
```

Arguments

`expr`	Input. An expression, expr_list, or `list()` of expressions. Also works with formulas or lists of formulas.
`...`	Alternating series of patterns and replacements, each a single expression (though alternatives can be specified with ?).
`patterns`	Patterns to look for. An expression, expr_list, or `list()` of expressions.
`replacements`	Replacements, one for each pattern.
`n`	Maximum number of times for each expression to make each replacement; default is `Inf`.
`env`	Environment for injections in `expr`, `pattern` (see expression).

Details

Patterns follow the syntax for `expr_match()`.

Value

The input expression(s) with any replacements made.

See Also

`expr_match()` to find patterns in expressions, and its cousins `expr_count()`, `expr_detect()`, `expr_extract()`, and `expr_locate()`.

Examples

```
# Example with alternating patterns and replacements
expr_replace({ 1 + 2 }, {1}, {one}, {2}, {two})

# Example with patterns and replacements in a list
expr_replace({ 1 + 2 }, patterns = expr_list({1}, {2}),
    replacements = expr_list({one}, {two}))

# Replace with captures
expr_replace({ 1 + 2 }, ~{ .A + .B }, { .A - .B })
```

`expr_sub` *Get or set a subexpression*

Description

These functions allow you to extract and/or modify a subexpression within an expression.

Usage

```
expr_sub(expr, idx, env = parent.frame())

expr_sub(expr, idx, env = parent.frame()) <- value
```

Arguments

`expr`	The expression to select from. Can also be a list of expressions, in which case the first element of `index` selects the expression from the list. Can also be a formula.
`idx`	A valid index: `NULL` or an integer vector.
`env`	Environment for any injections in `expr` (see expression).
`value`	Replacement; an expression.

Details

The `elixir` functions `expr_match()` and `expr_locate()` find matching "subexpressions" within expressions and return indices that allow accessing these subexpressions. For example, the expression `1 + 2 + 3` contains all the following subexpressions:

index	**subexpression**	**accessed with R code**
`NULL`	`1+2+3`	`expr`
`1`	`+`	`expr[[1]]`
`2`	`1+2`	`expr[[2]]`
`3`	`3`	`expr[[3]]`
`c(2,1)`	`+`	`expr[[2]][[1]]` or `expr[[c(2, 1)]]`
`c(2,2)`	`1`	`expr[[2]][[2]]` or `expr[[c(2, 2)]]`
`c(2,3)`	`2`	`expr[[2]][[3]]` or `expr[[c(2, 3)]]`

Any index returned by `expr_match()` or `expr_locate()` will either be `NULL` (meaning the whole expression / expression list) or an integer vector (e.g. `1` or `c(2,3)` in the table above).

Suppose you have an index, `idx`. If `idx` is an integer vector, you can just use `expr[[idx]]` to access the subexpression. But in the case where `idx` is `NULL`, R will complain that you are trying to select less than one element. The sole purpose of `expr_sub()` is to get around that issue and allow you to pass either `NULL` or an integer vector as the index you want for an expression or list of expressions.

Value

The element of the expression selected by `idx`.

See Also

`expr_match()`, `expr_locate()` which return indices to subexpressions.

Examples

```
expr = quote(y == a * x + b)
expr_sub(expr, NULL)
expr_sub(expr, 3)
expr_sub(expr, c(3, 3))

expr_sub(expr, c(3, 3)) <- quote(q)
print(expr)
```

lang2str — *Convert an expression into a string*

Description

The opposite of `str2lang()`, `lang2str()` converts an expression into a character string. Note that `lang2str()` does not support the normal expression syntax for `elixir`, so just expects an already-parsed expression.

Usage

```
lang2str(x)
```

Arguments

`x`	Expression to convert to a string.

Details

This function is essentially identical to `deparse1()`, which has been available since R 4.0.0, except with `collapse = ""` instead of `collapse = " "`.

Value

A character string suitable for printing.

Examples

```
lang2str(quote(a + b + c))
```

meld — *Code generation from template file*

Description

`meld` reads a specially-formatted file from filename `file` or as lines of text passed via unnamed arguments and returns these lines of text after performing substitutions of R code.

This function is experimental.

Usage

```
meld(
  ...,
  file = NULL,
  rules = NULL,
  reindent = TRUE,
  ipath = ".",
  env = rlang::env_clone(parent.frame())
)
```

Arguments

`...` Lines to be interpreted as the text. If there are any embedded newlines in a line, the line is split into multiple lines.

`file` File to be read in as the text.

`rules` Which rules to follow. You can pass a string from among `"C"`, `"C++"`, `"Lua"`, or `"R"`, or a list with elements:

- `comment` Character vector for comments (used when backticked lines are skipped); either NA for no comments, one string for end-of-line comments or two strings for delimited comments.
- `indent_more` Character vector of tokens which increase the indent level.
- `indent_less` Character vector of tokens which decrease the indent level.
- `indent_both` Character vector of tokens which decrease, then increase the indent level (see `reindent()`).
- `ignore` Comment and string literal delimiters (see `reindent()`).

If `NULL`, the default, either guess rules from the file extension, or if that is not possible, do not put in 'skipped' comments and do not reindent the result. `NA` to not try to guess.

`reindent` If `TRUE`, the default, reindent according to `rules`. If `FALSE`, do not reindent.

`ipath` Path to search for `#include`d files

`env` Environment in which to evaluate R expressions. The default is `rlang::env_clone(parent.frame())`, and it is best to clone the environment so that new declarations do not pollute the environment in question.

Details

As `meld` works through each line of the text, any blocks of text starting with the delimiter `/***R` and ending with `*/` are run as R code.

Outside these blocks, any substrings in the text delimited by `` `backticks` `` are interpreted as R expressions to be substituted into the line. If any of the backticked expressions are length 0, the line is commented out (with the message "[skipped]" appended) using the `comment` element of `rules`. If any of the backticked expressions are length L > 1, the entire interpreted line is repeated L times, separated by newlines and with elements of the expression in sequence.

There are some special sequences:

- `` `^expr` `` subs in `expr` only on the first line of a multi-line expansion

- `` `!^expr` `` subs in `expr` on all but the first line of a multi-line expansion
- `` `$expr` `` subs in `expr` only on the last line of a multi-line expansion
- `` `!$expr` `` subs in `expr` on all but the last line of a multi-line expansion
- `` `#include file` `` interprets `file` as an R expression resolving to a filename, runs that file through `meld`, and pastes in the result

The `#include` command must appear by itself on a line, and searches for files in the path `ipath`.

The function tries to guess `rules` from the file extension if that is possible. If the file extension is .c, then `"C"` is guessed; for .h, .hpp, or .cpp, `"C++"` is guessed; for .R, `"R"` is guessed; for .lua, `"Lua"` is guessed. Case is ignored for file extensions.

R blocks are evaluated immediately prior to the next-occurring backticked line, so variables modified in an R block are available to any backticked expression following the R block. Any remaining R blocks are run after remaining lines are interpreted.

If any line from the text ends with a single backslash `\`, the next line is concatenated to it. If any line from the text ends with a double backslash `\\`, the next line is concatenated to it with a newline as a separator. This allows backticked expressions to apply over multiple lines.

Value

The interpreted text as a single character string.

Examples

```
meld(
    "/***R",
    "names = c('a', 'b', 'c');",
    "dontdothis = NULL;",
    "*/",
    "double foo()",
    "{",
    "    double `names` = `1:3`;",
    "    double `dontdothis` = this_doesnt_matter;",
    "    return `paste(names, collapse = ' + ')`;",
    "}")
```

`reindent` *Reindent some lines of code*

Description

Using some fairly unsophisticated metrics, `reindent()` will take some lines of code and, according to its understanding of the rules for that language, reindent those lines. This is intended to help prettify automatically generated code.

This function is experimental.

Usage

```
reindent(lines, rules, tab = "    ", start = 0L)
```

Arguments

`lines`	Character vector with lines of text; can have internal newlines.
`rules`	Which rules to follow. You can pass a string from among `"C"`, `"C++"`, `"Lua"`, or `"R"`, or a list with elements: • `indent_more` Character vector of tokens which increase the indent level. • `indent_less` Character vector of tokens which decrease the indent level. • `indent_both` Character vector of tokens which decrease, then increase the indent level (see Details). • `ignore` Comment and string literal delimiters (see Details).
`tab`	Character string; what to use as an indent.
`start`	Indent level to start at.

Details

Conceptually, the function first ignores any comments or string literals. Then, line by line, `reindent` looks for tokens that signal either an increase in the indent level, a decrease in the indent level, or both at the same time. For example, in this Lua code:

```
if x == 1 then
    print 'one'
else
    print 'not one'
end
```

the `if` keyword increases the indent level, the `else` keyword both decreases and increases the indent level, and the `end` keyword decreases the indent level.

If provided, the `ignore` element of `rules` should be a list of character vectors. A character vector of length one is assumed to start a comment that runs to the end of the line (e.g. `"#"` in R). If length two, the two symbols are assumed to start and end a comment or string (e.g. `"/*"` and `"*/"` in C). If length three, then the first two symbols are start and end delimiters of a comment or string, while the third symbol is an "escape" character that escapes the end delimiter (and can also escape itself). This is typically a backslash.

`reindent()` supports "raw strings" in R, C, C++, and Lua code but only in limited cases. In R, when using raw character constants you must use an uppercase `R`, the double quote symbol and zero to two hyphens. In C/C++, when using raw string literals you must use the prefix `R`, and zero to two hyphens as the delimiter char sequence (plus parentheses). In Lua, you can use long brackets with zero to two equals signs. Any other attempt to use raw strings will probably break `reindent()`.

Other unusual character sequences may also break `reindent()`.

Value

Reindented lines as a character vector.

Examples

```
reindent(
    c(
        "if x == 1 then",
        "print 'one'",
        "else",
        "print 'not one'",
        "end"
    ),
    rules = "Lua")
```

---

translate *Translate an R expression*

Description

Takes an R expression (in the sense of `rlang::is_expression()`) and translates it into a character string giving the equivalent expression in another programming language, according to the supplied rules.

This function is experimental.

Usage

```
translate(expr, rules, env = parent.frame())
```

Arguments

`expr`	Expression or list of expressions to be translated.
`rules`	Which rules to follow. You can pass a string from among `"C"`, `"C++"`, `"Lua"`, or `"R"`, or a list with translation rules (see Details).
`env`	Environment for injections in `expr` (see expression).

Details

The parameter `rules` can be a character string naming a "built-in" ruleset. Otherwise, `rules` should be a list with the following elements:

- `ops`: an unnamed list of operator definitions, each of which should be a list with four elements:
  - `arity` the number of operands
  - `prec` the precedence of the operator (lower numbers equal higher precedence)
  - `assoc` the associativity of the operator, either `"LTR"`, `"RTL"`, or anything else for no associativity
  - `str` a `glue::glue()` format string with `{A[1]}`, `{A[2]}`, etc., standing in for the first, second, etc. operands.

- nopar a numeric vector with indices of arguments to the operator which should never be enclosed in parentheses. The default and usual value is integer(0), but (for example) it can be 2 for the [operator, as parentheses within the second argument (the content of the brackets) are redundant.

  The function `elixir:::op` can help to assemble such lists.

- `paren` a `glue::glue()` format string with `{x}` standing in for the enclosed expression. Describes how parentheses are expressed in the target language. Example: `"({x})"` is correct for virtually all programming languages.
- `symbol`: a function which takes a symbol and returns a character string, representing the name of that symbol in the target language. This could just be equal to base::as.character, but it can be changed to something else in case you want name mangling, or e.g. some processing to replace `.` in symbols with some other character (as `.` are often not allowed as part of symbols in popular languages).
- `literal`: a named list in which the name refers to the class of the operand to translate, and the value should be a function of a single argument (the operand) returning a character string.

It may be helpful to inspect `elixir:::ruleset` to clarify the above format.

There are some important shortcomings to `translate()`. Here are some potential pitfalls:

- Named arguments are not supported, because we cannot translate an R function call like `mean(x, na.rm = TRUE)` without knowing which parameter of `mean` matches to `na.rm`.
- Division: An R expression like `1/3` gets translated into `1./3.` in C/C++, as numeric literals are coerced to type `double`. So both of these evaluate to 0.333. However, the R expression `1L/3L` will get translated into `1/3` in C/C++, which evaluates to 0 (as it is integer division).
- Modulo: R uses "Knuth's modulo", where `a %% b` has the same sign as `b`. Lua also uses Knuth's modulo, but C/C++ use "truncated modulo", where `a % b` has the same sign as `a`. (see Wikipedia for details). So when converting a modulo expression from R to C/C++, a call to the function `kmod` is generated in the C/C++ expression. This is not a standard library function, so you will have to provide a definition yourself. A workable definition is: `double kmod(double x, double y) { double r = fmod(x, y); return r && r < 0 != y < 0 ? r + y : r; }`
- Types: In R, the type of `a %% b` and of `a %/% b` depends on the type of `a` and `b` (if both are integers, the result is an integer; if at least one is numeric, the result is numeric).
- Chained assignment does not work in Lua.

Value

The translated expression as a single character string.

Examples

```
translate({x ^ y}, "C++")
```

Index