

Package ‘astgrepr’

June 4, 2025

Type Package

Title Parse and Manipulate R Code

Version 0.1.0

Description Parsing R code is key to build tools such as linters and stylers.
This package provides a binding to the 'Rust' crate 'ast-grep' so that one
can parse and explore R code.

License MIT + file LICENSE

Encoding UTF-8

Config/rextendr/version 0.4.0.9000

RoxygenNote 7.3.2

Depends R (>= 4.2)

Imports checkmate, rraply, stats, yaml

Suggests knitr, rmarkdown, rstudioapi, spelling, tinytest

URL <https://github.com/etiennebacher/astgrepr>,
<https://astgrepr.etiennebacher.com/>

BugReports <https://github.com/etiennebacher/astgrepr/issues>

SystemRequirements Cargo (Rust's package manager), rustc (>= 1.78.0)

VignetteBuilder knitr

Language en-US

NeedsCompilation yes

Author Etienne Bacher [aut, cre, cph]

Maintainer Etienne Bacher <etienne.bacher@protonmail.com>

Repository CRAN

Date/Publication 2025-06-04 12:20:06 UTC

Contents

| | |
|---------------------------|-----------|
| ast_rule | 2 |
| node-find | 4 |
| node-fix | 6 |
| node-get-match | 7 |
| node-info | 9 |
| node-is | 10 |
| node-range | 11 |
| node-text | 12 |
| node-traversal | 13 |
| node_get_root | 15 |
| node_kind | 16 |
| pattern_rule | 16 |
| relational_rule | 17 |
| tree_new | 18 |
| tree_rewrite | 18 |
| tree_root | 20 |
| Index | 21 |

| | |
|----------|---------------------|
| ast_rule | <i>Build a rule</i> |
|----------|---------------------|

Description

Rules are the core of astgrepr. Those are used to search for nodes and are used in node_match*() and node_find*() functions. ast_rule() is a very flexible function that allows one to build simple rules but also much more complex and specific ones.

Usage

```
ast_rule(  
    pattern = NULL,  
    kind = NULL,  
    regex = NULL,  
    inside = NULL,  
    has = NULL,  
    precedes = NULL,  
    follows = NULL,  
    all = NULL,  
    any = NULL,  
    not = NULL,  
    matches = NULL,  
    id = NULL  
)
```

Arguments

| | |
|----------|--|
| pattern | The pattern to look for. This can be a string or an object of class "astgrep_pattern_rule" created by <code>pattern_rule()</code> . This can contain meta-variables to capture certain elements. Those meta-variables can then be recovered with <code>node_get_match()</code> and <code>node_get_multiple_matches()</code> . The meta-variables must start with \$ and have only uppercase letters, e.g. \$VAR. |
| kind | The kind of nodes to look for. |
| regex | A regex used to look for nodes. This must follow the syntax of the Rust regex crate . |
| inside | In which node should the node we look for be positioned? This can be another rule made with <code>ast_rule()</code> or an object of class "astgrep_relational_rule" created with <code>relational_rule()</code> . |
| has | Same input type as inside, but this looks for nodes that contain another type of node. |
| precedes | Same input type as inside, but this looks for nodes that precede another type of node. |
| follows | Same input type as inside, but this looks for node that follow another type of node. |
| all | This takes one or a list of rules made with <code>ast_rule()</code> . It only matches nodes that respect all of the rules. |
| any | This takes one or a list of rules made with <code>ast_rule()</code> . It matches nodes that respect any of the rules. |
| not | This takes one or a list of rules made with <code>ast_rule()</code> . It excludes those nodes from the selection. |
| matches | This takes the id of another rule. It is useful to reuse rules. |
| id | The name of this rule. This can be reused in another rule with matches. |

Value

A list (possibly nested) with the class "astgrep_rule".

About meta-variables

Meta-variables allow us to capture some of the content in a pattern. Usually, using \$ followed by an id in uppercase letters is enough:

```
src <- "any(duplicated(x))"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(pattern = "any(duplicated($A))"))
#> <List of 1 rule>
#> |--rule_1: 1 node
```

However, in some cases using `$` is a problem. For instance, if we want to capture a column name coming after `$`, then we can't use `$` both as code and as identifier.

```
src <- "df$a"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(pattern = "df$$A"))
#> <List of 1 rule>
#> |--rule_1: 0 node
```

In this situation, we can use `μ` instead:

```
root |>
  node_find(ast_rule(pattern = "df$μA"))
#> <List of 1 rule>
#> |--rule_1: 1 node
```

Examples

```
ast_rule(pattern = "print($A)")

ast_rule(
  pattern = "print($A)",
  inside = ast_rule(
    any = ast_rule(
      kind = c("for_statement", "while_statement")
    )
  )
)
```

node-find

Find node(s) matching a pattern

Description

Those functions find one or several nodes based on some rule:

- `node_find()` returns the first node that is found;
- `node_find_all()` returns a list of all nodes found.

Some arguments (such as `kind`) require some knowledge of the tree-sitter grammar of R. This grammar can be found here: <https://github.com/r-lib/tree-sitter-r/blob/main/src/grammar.json>.

Usage

```
node_find(x, ..., files = NULL)
```

```
node_find_all(x, ..., files = NULL)
```

Arguments

| | |
|--------------------|---|
| <code>x</code> | A node, either from <code>tree_root()</code> or from another <code>node_*</code> () function. |
| <code>...</code> | Any number of rules created with <code>ast_rule()</code> . |
| <code>files</code> | A vector of filenames containing rules. Those must be <code>.yaml</code> files. |

Value

`node_find()` returns a single `SgNode`.

`node_find_all()` returns a list of `SgNodes`.

Examples

```
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(mtcars)
      any(duplicated(x))"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(pattern = "any(duplicated($A))"))

root |>
  node_find_all(ast_rule(pattern = "any(duplicated($A))"))

# using the 'kind' of the nodes to find elements
src <- "
  a <- 1
  while (TRUE) { print('a') }
"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(kind = "while_statement"))

# one can pass several rules at once
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(mtcars)"
```

```

    any(duplicated(x))
    while (TRUE) { print('a') }"
root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(
    ast_rule(pattern = "any(duplicated($A))"),
    ast_rule(kind = "while_statement")
  )

root |>
  node_find_all(
    ast_rule(pattern = "any(duplicated($A))"),
    ast_rule(kind = "while_statement")
  )

```

node-fix

*Change the code in the tree***Description**

`node_replace()` gives the replacement for a particular node. `node_replace_all()` does the same but for several nodes (e.g. the output of `node_find_all()`). The output of those functions can be passed to `tree_rewrite()` to rewrite the entire input code with those replacements.

Usage

```
node_replace(x, ...)
```

```
node_replace_all(x, ...)
```

Arguments

| | |
|------------------|---|
| <code>x</code> | A node, either from <code>tree_root()</code> or from another <code>node_*</code> () function. |
| <code>...</code> | Named elements where the name is a rule ID and the value is a character string indicating the replacement to apply to nodes that match this rule. Meta-variables are accepted but the syntax is different: they must be wrapped in <code>~~</code> , e.g. <code>"anyNA(~~VAR~~)"</code> . |

Value

A list where each element is the replacement for a piece of the code. Each element is a list containing 3 sub-elements:

- the start position for the replacement
- the end position for the replacement
- the text used as replacement

Examples

```

src <- "
x <- c(1, 2, 3)
any(duplicated(x), na.rm = TRUE)
any(duplicated(x))
if (any(is.na(x))) {
  TRUE
}
any(is.na(y))"

root <- tree_new(src) |>
  tree_root()

### Only replace the first nodes found by each rule

nodes_to_replace <- root |>
  node_find(
    ast_rule(id = "any_na", pattern = "any(is.na($VAR))"),
    ast_rule(id = "any_dup", pattern = "any(duplicated($VAR))")
  )

nodes_to_replace |>
  node_replace(
    any_na = "anyNA(~VAR~)",
    any_dup = "anyDuplicated(~VAR~) > 0"
  )

### Replace all nodes found by each rule

nodes_to_replace <- root |>
  node_find(
    ast_rule(id = "any_na", pattern = "any(is.na($VAR))"),
    ast_rule(id = "any_dup", pattern = "any(duplicated($VAR))")
  )

nodes_to_replace |>
  node_replace(
    any_na = "anyNA(~VAR~)",
    any_dup = "anyDuplicated(~VAR~) > 0"
  )

```

node-get-match

Get the match(es) from a meta-variable

Description

Those functions extract the content of the meta-variable specified in `node_find()`:

- `node_get_match()` is used when the meta-variable refers to a single pattern, e.g. `plot($A)`;

- `node_get_multiple_matches()` is used when the meta-variable captures all elements in a pattern, e.g. `"plot($$$A)"`.

Usage

```
node_get_match(x, meta_var)
```

```
node_get_multiple_matches(x, meta_var)
```

Arguments

| | |
|-----------------------|---|
| <code>x</code> | A node, either from <code>tree_root()</code> or from another <code>node_*</code> () function. |
| <code>meta_var</code> | The name given to one of the meta-variable(s) in <code>node_find()</code> . |

Value

`node_get_match()` returns a list of depth 1, where each element is the node corresponding to the rule passed (this can be of length 0 if no node is matched). `node_get_multiple_matches()` also returns a list of depth 1, but each element can contain multiple nodes when the meta-variable captures all elements in a pattern.

Examples

```
src <- "x <- rnorm(100, mean = 2)
      plot(mtcars)"

root <- src |>
      tree_new() |>
      tree_root()

# we capture a single element with "$A" so node_get_match() can be used
root |>
  node_find(ast_rule(pattern = "plot($A)")) |>
  node_get_match("A")

# we can specify the variable to extract
root |>
  node_find(ast_rule(pattern = "rnorm($A, $B)")) |>
  node_get_match("B")

# we capture many elements with "$$$A" so node_get_multiple_matches() can
# be used here
root |>
  node_find(ast_rule(pattern = "rnorm($$$A)")) |>
  node_get_multiple_matches("A")
```


node-info

*Get more precise information on a node***Description**

Get more precise information on a node

Usage

```
node_matches(x, ..., files = NULL)

node_inside(x, ..., files = NULL)

node_has(x, ..., files = NULL)

node_precedes(x, ..., files = NULL)

node_follows(x, ..., files = NULL)
```

Arguments

| | |
|--------------------|---|
| <code>x</code> | A node, either from <code>tree_root()</code> or from another <code>node_*</code> () function. |
| <code>...</code> | Any number of rules created with <code>ast_rule()</code> . |
| <code>files</code> | A vector of filenames containing rules. Those must be <code>.yaml</code> files. |

Value

A list containing as many elements as there are nodes as input.

Examples

```
src <- "
print('hi')
fn <- function() {
  print('hello')
}
"

root <- src |>
  tree_new() |>
  tree_root()

some_node <- root |>
  node_find(ast_rule(pattern = "print($A)"))

node_text(some_node)

# Check if a node matches a specific rule
some_node |>
```

```

node_get_match("A") |>
node_matches(ast_rule(kind = "argument"))

# Check if a node is inside another one
some_node |>
  node_get_match("A") |>
  node_inside(ast_rule(kind = "call"))

```

node-is

Get information on nodes

Description

Get information on whether a node is a leaf (meaning that it doesn't have any children) and whether it is named.

Usage

```

node_is_leaf(x)

node_is_named(x)

node_is_named_leaf(x)

```

Arguments

x A node, either from `tree_root()` or from another `node_*`() function.

Value

A logical value.

Examples

```

src <- "x <- rnorm(100, mean = 2)
  any(duplicated(y))
  x <- z + 1
  any(duplicated(x))"

root <- src |>
  tree_new() |>
  tree_root()

node_is_leaf(root)

root |>
  node_find(ast_rule(pattern = "z")) |>
  node_is_leaf()

```

```

root |>
  node_find(ast_rule(pattern = "z")) |>
  node_is_named()

```

node-range

Get the start and end positions of a node

Description

Get the start and end positions of a node

Usage

```

node_range(x)

node_range_all(x)

```

Arguments

x A node, either from `tree_root()` or from another `node_*`() function.

Value

A list of two elements: `start` and `end`. Each of those is a vector with two values indicating the row and column. Those are 0-indexed.

Examples

```

src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(x)
      any(duplicated(x))"

root <- src |>
  tree_new() |>
  tree_root()

node_range(root)

root |>
  node_find(ast_rule(pattern = "rnorm($$A)")) |>
  node_range()

# There is also an "_all" variant when there are several nodes per rule
root |>
  node_find_all(
    ast_rule(pattern = "any(duplicated($A))"),
    ast_rule(pattern = "plot($A)")
  ) |>
  node_range_all()

```

node-text

Extract the code corresponding to one or several nodes

Description

Those functions extract the code corresponding to the node(s):

- `node_text()` applies on a single node, for example the output of `node_get_match()`
- `node_text_all()` applies on a list of nodes, for example the output of `node_get_multiple_matches()`

Usage

```
node_text(x)
```

```
node_text_all(x)
```

Arguments

`x` A node, either from `tree_root()` or from another `node_*`() function.

Value

A list with as many elements as there are in the input. Each element is a list itself with the text corresponding to the input.

Examples

```
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(mtcars)
      any(duplicated(x))"

root <- src |>
  tree_new() |>
  tree_root()

# node_text() must be applied on single nodes
root |>
  node_find(ast_rule(pattern = "plot($A)")) |>
  node_text()

# node_find_all() returns a list on nodes on which
# we can use node_text_all()
root |>
  node_find_all(ast_rule(pattern = "any(duplicated($A))")) |>
  node_text_all()
```

node-traversal

*Navigate the tree***Description**

This is a collection of functions used to navigate the tree. Some of them have a variant that applies on a single node (e.g. `node_next()`) and one that applies on a list of nodes (e.g. `node_next_all()`):

- `node_prev()`, `node_prev_all()`, `node_next()`, and `node_next_all()` get the previous and next node(s) that are at the same depth as the current node;
- `node_parent()`, `node_ancestors()`, `node_child()` and `node_children()` get the node(s) that are above or below the current node in terms of depth. All nodes except the root node have at least one node (the root).

Usage

```
node_parent(x)
```

```
node_child(x, nth)
```

```
node_ancestors(x)
```

```
node_children(x)
```

```
node_next(x)
```

```
node_next_all(x)
```

```
node_prev(x)
```

```
node_prev_all(x)
```

Arguments

| | |
|------------------|---|
| <code>x</code> | A node, either from <code>tree_root()</code> or from another <code>node_*</code> () function. |
| <code>nth</code> | Integer. The child node to find. This is 0-indexed, so setting <code>nth = 0</code> gets the first child. |

Value

A node

Examples

```
### get the previous/next node -----
src <- "
```

```

print('hi there')
a <- 1
fn <- function(x) {
  x + 1
}
"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(pattern = "a <- $A")) |>
  node_prev() |>
  node_text()

root |>
  node_find(ast_rule(pattern = "a <- $A")) |>
  node_next() |>
  node_text()

# there are nodes inside the function, but there are no more nodes on the
# same level as "fn"
root |>
  node_find(ast_rule(pattern = "a <- $A")) |>
  node_next_all() |>
  node_text_all()

### get the parent/child node -----

src <- "
print('hi there')
a <- 1
fn <- function(x) {
  x + 1
}
"

root <- src |>
  tree_new() |>
  tree_root()

root |>
  node_find(ast_rule(pattern = "$VAR + 1")) |>
  node_parent() |>
  node_text()

root |>
  node_find(ast_rule(pattern = "$VAR + 1")) |>
  node_ancestors() |>
  node_text_all()

root |>
  node_find(ast_rule(pattern = "$VAR + 1")) |>

```

```

node_child(0) |>
node_text()

root |>
node_find(ast_rule(pattern = "$VAR + 1")) |>
node_children() |>
node_text_all()

```

node_get_root

Recover the tree root from a node

Description

Recover the tree root from a node

Usage

```
node_get_root(x)
```

Arguments

x A node, either from `tree_root()` or from another `node_*`() function.

Value

A list of two elements: start and end. Each of those is a vector with two values indicating the row and column. Those are 0-indexed.

Examples

```

src <- "
print('hi')
fn <- function() {
  print('hello')
}
"

root <- src |>
tree_new() |>
tree_root()

root |>
node_find(ast_rule(pattern = "print($A)")) |>
node_get_root() |>
tree_root() |>
node_text()

```

| | |
|-----------|--------------------------------|
| node_kind | <i>Find the kind of a node</i> |
|-----------|--------------------------------|

Description

Find the kind of a node

Usage

```
node_kind(x)
```

Arguments

x A node, either from `tree_root()` or from another `node_*`() function.

Value

A list with as many elements as in the input. Each element is a character value.

Examples

```
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      x <- z + 1
      any(duplicated(x))"

root <- src |>
      tree_new() |>
      tree_root()

root |>
  node_find(ast_rule(pattern = "any(duplicated($VAR))")) |>
  node_kind()

root |>
  node_find(ast_rule(pattern = "$X + $VALUE")) |>
  node_kind()
```

| | |
|--------------|-----------------------------|
| pattern_rule | <i>Build a pattern rule</i> |
|--------------|-----------------------------|

Description

This is a specific type of rule. It can be used in the more general ruleset built with `ast_rule()`.

Usage

```
pattern_rule(selector = NULL, context = NULL, strictness = "smart")
```


Arguments

| | |
|------------|---|
| selector | Defines the surrounding code that helps to resolve any ambiguity in the syntax. |
| context | Defines the sub-syntax node kind that is the actual matcher of the pattern. |
| strictness | Optional, defines how strictly pattern will match against nodes. See 'Details'. |

Details

The strictness parameter defines the type of nodes the ast-grep matcher should consider. It has the following values:

- `cst`: All nodes in the pattern and target code must be matched. No node is skipped.
- `smart`: All nodes in the pattern must be matched, but it will skip unnamed nodes in target code. This is the default behavior.
- `ast`: Only named AST nodes in both pattern and target code are matched. All unnamed nodes are skipped.
- `relaxed`: Named AST nodes in both pattern and target code are matched. Comments and unnamed nodes are ignored.
- `signature`: Only named AST nodes' kinds are matched. Comments, unnamed nodes and text are ignored.

More information: <https://ast-grep.github.io/guide/rule-config/atomic-rule.html#pattern-object>

Value

An list of class `astgrep_pattern_rule`

| | |
|------------------------------|--------------------------------|
| <code>relational_rule</code> | <i>Build a relational rule</i> |
|------------------------------|--------------------------------|

Description

Build a relational rule

Usage

```
relational_rule(stopBy = "neighbor", field = NULL, regex = NULL)
```

Arguments

| | |
|--------|------|
| stopBy | todo |
| field | todo |
| regex | todo |

Value

An list of class `astgrep_relational_rule`

| | |
|----------|-----------------------------|
| tree_new | <i>Create a syntax tree</i> |
|----------|-----------------------------|

Description

This function takes R code as string and creates the corresponding abstract syntax tree (AST) from which we can query nodes.

Usage

```
tree_new(txt, file, ignore_tags = "ast-grep-ignore")
```

Arguments

| | |
|-------------|---|
| txt | A character string of length 1 containing the code to parse. If provided, file must not be provided. |
| file | Path to file containing the code to parse. If provided, txt must not be provided. |
| ignore_tags | Character vector indicating the tags to ignore. Default is "ast-grep-ignore", meaning that any line that follows # ast-grep-ignore will be ignored in the output of node_*() functions. |

Value

An abstract syntax tree containing nodes

Examples

```
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(x)
      any(duplicated(x))"

tree_new(src)
```

| | |
|--------------|---|
| tree_rewrite | <i>Rewrite the tree with a list of replacements</i> |
|--------------|---|

Description

Rewrite the tree with a list of replacements

Usage

```
tree_rewrite(root, replacements)
```

Arguments

root The root tree, obtained via `tree_root()`

replacements A list of replacements, obtained via `node_replace()` or `node_replace_all()`.

Value

A string character corresponding to the code used to build the tree root but with replacements applied.

Examples

```
src <- "x <- c(1, 2, 3)
any(duplicated(x), na.rm = TRUE)
any(duplicated(x))
if (any(is.na(x))) {
  TRUE
}
any(is.na(y))"

root <- tree_new(src) |>
  tree_root()

### Only replace the first nodes found by each rule

nodes_to_replace <- root |>
  node_find(
    ast_rule(id = "any_na", pattern = "any(is.na($VAR))"),
    ast_rule(id = "any_dup", pattern = "any(duplicated($VAR))")
  )

fixes <- nodes_to_replace |>
  node_replace(
    any_na = "anyNA(~VAR~)",
    any_dup = "anyDuplicated(~VAR~) > 0"
  )

# original code
cat(src)

# new code
tree_rewrite(root, fixes)

### Replace all nodes found by each rule

nodes_to_replace <- root |>
  node_find_all(
    ast_rule(id = "any_na", pattern = "any(is.na($VAR))"),
    ast_rule(id = "any_dup", pattern = "any(duplicated($VAR))")
  )
```

```
fixes <- nodes_to_replace |>
  node_replace_all(
    any_na = "anyNA(~~VAR~~)",
    any_dup = "anyDuplicated(~~VAR~~) > 0"
  )

# original code
cat(src)

# new code
tree_rewrite(root, fixes)
```

tree_root*Get the root of the syntax tree*

Description

This function takes a tree created by `tree_new()` and returns the root node containing all subsequent nodes.

Usage

```
tree_root(x)
```

Arguments

x A tree created by `tree_new()`.

Value

A node corresponding to the root of the abstract syntax tree

Examples

```
src <- "x <- rnorm(100, mean = 2)
      any(duplicated(y))
      plot(x)
      any(duplicated(x))"

tree <- tree_new(src)
tree_root(tree)
```

Index

ast_rule, 2

node-find, 4

node-fix, 6

node-get-match, 7

node-info, 9

node-is, 10

node-range, 11

node-text, 12

node-traversal, 13

node_ancestors (node-traversal), 13

node_child (node-traversal), 13

node_children (node-traversal), 13

node_find (node-find), 4

node_find(), 7

node_find_all (node-find), 4

node_follows (node-info), 9

node_get_match (node-get-match), 7

node_get_match(), 3, 12

node_get_multiple_matches
(node-get-match), 7

node_get_multiple_matches(), 3, 12

node_get_root, 15

node_has (node-info), 9

node_inside (node-info), 9

node_is_leaf (node-is), 10

node_is_named (node-is), 10

node_is_named_leaf (node-is), 10

node_kind, 16

node_matches (node-info), 9

node_next (node-traversal), 13

node_next_all (node-traversal), 13

node_parent (node-traversal), 13

node_precedes (node-info), 9

node_prev (node-traversal), 13

node_prev_all (node-traversal), 13

node_range (node-range), 11

node_range_all (node-range), 11

node_replace (node-fix), 6

node_replace_all (node-fix), 6

node_text (node-text), 12

node_text_all (node-text), 12

pattern_rule, 16

relational_rule, 17

tree_new, 18

tree_new(), 20

tree_rewrite, 18

tree_root, 20

tree_root(), 5, 6, 8–13, 15, 16