Package 'bonsaisitter'

September 17, 2026

Type Package

Title Bindings to the 'Tree-Sitter' Parsing Library

Version 0.1.2

Date 2026-09-08

Description A runtime for the 'Tree-sitter' parsing library <https://tree-sitter.github.io/tree-sitter/> that mirrors the API of the 'treesitter' package, so it can serve as a drop-in replacement. Parses source code into concrete syntax trees and updates them incrementally as the source changes. Grammars are supplied by separate packages such as 'treesitter.r', so the runtime itself depends on nothing beyond base R.

License MIT + file LICENSE

Copyright cornball.ai, except the bundled tree-sitter sources under src/tree-sitter, whose holders are listed in inst/COPYRIGHTS.

URL https://github.com/cornball-ai/bonsaisitter

BugReports https://github.com/cornball-ai/bonsaisitter/issues

Encoding UTF-8

NeedsCompilation yes

Suggests tinytest, treesitter, treesitter.r

Author Troy Hernandez [aut, cre] (ORCID: <https://orcid.org/0009-0005-4248-604X>), cornball.ai [cph], Max Brunsfeld [ctb, cph] (tree-sitter C library, bundled under src/tree-sitter), International Business Machines Corporation [cph] (ICU UTF-8 and UTF-16 headers, src/tree-sitter/lib/src/unicode), Unicode, Inc. [cph] (ICU license, src/tree-sitter/lib/src/unicode/LICENSE), Mathias Panzenböck [ctb] (src/tree-sitter/lib/src/portable/endian.h, public domain)

Maintainer Troy Hernandez <troy@cornball.ai>

Repository CRAN

Date/Publication 2026-09-17 09:00:09 UTC

Contents

`as.data.frame.tree_sitter_node` *Convert a node's descendants to a data frame*

Description

Convert a node's descendants to a data frame

Usage

```
## S3 method for class 'tree_sitter_node'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

`x`	A `tree_sitter_node`.
`row.names`	Ignored.
`optional`	Ignored.
`...`	Ignored.

Value

A base data frame with columns: type, named, text, start_row, start_col, end_row, end_col, start_byte, end_byte.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- f(1)", treesitter.r::language()))
  as.data.frame(node)
}
```

`audit_translation` *Audit a code translation for drifted numeric constants*

Description

Compares a reference scope (e.g. the original Python or torch code) with its port, reporting numeric literals present in one but not the other. Parity tests prove the paths they exercise; this catches the drift they do not – wrong constants, dropped operations, unported branches. Call names are returned too for eyeballing with a noise filter.

Usage

```
audit_translation(reference, port, lang = "r", normalize = TRUE)
```

Arguments

`reference`	Character scalar of reference source.
`port`	Character scalar of the port's source.
`lang`	Language of both (`"r"`, `"python"`, `"cpp"`, `"go"`, `"rust"`, `"javascript"`). Pass a length-2 vector to parse reference and port as different languages (e.g. `c("python", "r")`).
`normalize`	Logical; strip `L` suffixes and coerce so `4.0 == 4L == 4`.

Details

Numeric literals are normalized by default so `4.0`, `4L`, and `4` compare equal.

Value

A list: `literals_missing` (in reference, not port), `literals_extra` (in port, not reference), and the full `reference` / `port` extractions.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  audit_translation("clamp(x, 1e-10); y * 8", "pmax(x, 1e-10); y * 8")
}
```

is_language *Is* x *a language?*

Description

Is x a language?

Usage

```
is_language(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  is_language(treesitter.r::language())
}
```

is_node *Is* x *a node?*

Description

Is x a node?

Usage

```
is_node(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x + 1", treesitter.r::language()))
  c(is_node(node), is_node("x + 1"))
}
```

---

is_parser *Is* x *a parser?*

Description

Is x a parser?

Usage

```
is_parser(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  is_parser(parser(treesitter.r::language()))
}
```

---

is_query *Is* x *a query?*

Description

Is x a query?

Usage

```
is_query(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  is_query(query(treesitter.r::language(), "(identifier) @id"))
}
```

---

is_tree *Is* x *a tree?*

Description

Is x a tree?

Usage

```
is_tree(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  is_tree(text_parse("x <- 1", treesitter.r::language()))
}
```

---

is_tree_cursor *Is* x *a tree cursor?*

Description

Is x a tree cursor?

Usage

```
is_tree_cursor(x)
```

Arguments

x	An object.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  is_tree_cursor(tree_walk(text_parse("x <- 1", treesitter.r::language())))
}
```

---

language-fields *Language fields*

Description

- language_field_count() is the number of distinct fields. - language_field_name_for_id() maps a field id to its name. - language_field_id_for_name() maps a field name to its id.

Usage

```
language_field_count(x)

language_field_name_for_id(x, id)

language_field_id_for_name(x, name)
```

Arguments

x	A tree_sitter_language.
id	A field id.
name	A field name.

Value

Counts and ids as a double; names as a string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  language_field_count(lang)
  language_field_name_for_id(lang, language_field_id_for_name(lang, "lhs"))
}
```

language-states *Language parse states*

Description

Language parse states

Usage

```
language_state_count(x)

language_next_state(x, state, symbol)
```

Arguments

x	A tree_sitter_language.
state	A parse state id.
symbol	A symbol id.

Value

A single double.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  language_state_count(lang)
  node <- tree_root_node(text_parse("x <- 1", lang))
  language_next_state(lang, node_parse_state(node), node_grammar_symbol(node))
}
```

language-symbols *Language symbols*

Description

- language_symbol_count() is the number of distinct node types. - language_symbol_name() maps a symbol id to its name. - language_symbol_for_name() maps a name to its symbol id.

Usage

```
language_symbol_count(x)

language_symbol_name(x, symbol)

language_symbol_for_name(x, name, named = TRUE)
```

Arguments

x	A tree_sitter_language.
symbol	A symbol id.
name	A node type name.
named	Whether to look up the named or anonymous variant.

Value

Counts and ids as a double; names as a string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  language_symbol_count(lang)
  language_symbol_name(lang, language_symbol_for_name(lang, "identifier"))
}
```

---

language_name *The name of a language*

Description

The name of a language

Usage

```
language_name(x)
```

Arguments

x	A tree_sitter_language.

Value

A single string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  language_name(treesitter.r::language())
}
```

literals_and_calls *Extract numeric literals and call names from source code*

Description

Parses code with tree-sitter and walks the AST collecting every numeric literal (integer / float) and the callee name of every call. The building block for audit_translation. Requires the grammar package for lang (treesitter.r, treesitter.python, treesitter.cpp, treesitter.go, treesitter.rust, or treesitter.javascript) to be installed.

Usage

```
literals_and_calls(
  code,
  lang = c("r", "python", "cpp", "go", "rust", "javascript")
)
```

Arguments

code	Character scalar of source code.
lang	Language: "r" (default), "python", "cpp", "go", "rust", or "javascript".

Value

A list with literals and calls (character vectors, in source order).

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  literals_and_calls("f(1L, 2.5) + g(3)")
}
```

node-bytes *A node's start / end byte*

Description

A node's start / end byte

Usage

```
node_start_byte(x)

node_end_byte(x)
```

Arguments

x	A tree_sitter_node.

Value

A single double (0-indexed).

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  c(node_start_byte(node), node_end_byte(node))
}
```

---

node-child *Access a child by index (1-indexed)*

Description

Access a child by index (1-indexed)

Usage

```
node_child(x, i)

node_named_child(x, i)
```

Arguments

x	A tree_sitter_node.
i	A 1-indexed child position.

Value

A tree_sitter_node, or NULL.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  c(node_type(node_child(expr, 2)), node_type(node_named_child(expr, 2)))
}
```

node-child-count *Child counts*

Description

Child counts

Usage

```
node_child_count(x)

node_named_child_count(x)
```

Arguments

x	A tree_sitter_node.

Value

A single double.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  c(node_child_count(expr), node_named_child_count(expr))
}
```

node-children *All children of a node*

Description

All children of a node

Usage

```
node_children(x)

node_named_children(x)
```

Arguments

x	A tree_sitter_node.

Value

A list of `tree_sitter_nodes`.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  vapply(node_children(expr), node_type, character(1))
  vapply(node_named_children(expr), node_type, character(1))
}
```

`node-field-name` *Field name for a child (1-indexed)*

Description

Field name for a child (1-indexed)

Usage

```
node_field_name_for_child(x, i)

node_field_name_for_named_child(x, i)
```

Arguments

`x`	A `tree_sitter_node`.
`i`	A 1-indexed child position.

Value

A single string, or `NA` if the child has no field.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  c(node_field_name_for_child(expr, 1), node_field_name_for_named_child(expr, 2))
}
```

node-locate *Locate nodes by byte or point*

Description

Locate nodes by byte or point

Usage

```
node_first_child_for_byte(x, byte)

node_first_named_child_for_byte(x, byte)

node_descendant_for_byte_range(x, start, end)

node_named_descendant_for_byte_range(x, start, end)

node_descendant_for_point_range(x, start_point, end_point)

node_named_descendant_for_point_range(x, start_point, end_point)
```

Arguments

x	A tree_sitter_node.
byte, start, end	0-indexed byte offsets.
start_point, end_point	tree_sitter_point objects.

Value

A tree_sitter_node, or NULL.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  root <- tree_root_node(text_parse("x <- 1\ny <- 2", treesitter.r::language()))
  node_text(node_first_named_child_for_byte(root, 7))
  node_text(node_descendant_for_byte_range(root, 7, 7))
  node_text(node_descendant_for_point_range(root, point(1, 0), point(1, 0)))
}
```

node-navigation *Sibling and parent navigation*

Description

Sibling and parent navigation

Usage

```
node_parent(x)

node_next_sibling(x)

node_previous_sibling(x)

node_next_named_sibling(x)

node_previous_named_sibling(x)
```

Arguments

x	A tree_sitter_node.

Value

A tree_sitter_node, or NULL.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  lhs <- node_child_by_field_name(node_child(tree_root_node(tree), 1), "lhs")
  c(node_type(node_parent(lhs)), node_type(node_next_sibling(lhs)),
    node_type(node_next_named_sibling(lhs)))
}
```

node-points *A node's start / end point*

Description

A node's start / end point

Usage

```
node_start_point(x)

node_end_point(x)
```

Arguments

x	A tree_sitter_node.

Value

A tree_sitter_point.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1\ny <- 2", treesitter.r::language()))
  node_start_point(node)
  node_end_point(node)
}
```

---

node-state *Node error and state predicates*

Description

- node_has_error() is TRUE if the node or any descendant is an error or is missing. - node_is_error() is TRUE if the node itself is a syntax error. - node_is_missing() is TRUE if the node is inserted by the parser to recover from an error. - node_is_extra() is TRUE if the node is "extra" (e.g. a comment).

Usage

```
node_has_error(x)

node_is_error(x)

node_is_missing(x)

node_is_extra(x)
```

Arguments

x	A tree_sitter_node.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  ok <- tree_root_node(text_parse("x <- 1", lang))
  broken <- tree_root_node(text_parse("x <- ", lang))
  c(node_has_error(ok), node_has_error(broken))
}
```

`node-symbols` *Node symbols and parse states*

Description

Node symbols and parse states

Usage

```
node_symbol(x)

node_grammar_symbol(x)

node_parse_state(x)

node_next_parse_state(x)
```

Arguments

x	A `tree_sitter_node`.

Value

A single double.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_symbol(node)
  language_symbol_name(node_language(node), node_symbol(node))
  node_parse_state(node)
}
```

node_child_by_field_id *A child by field id*

Description

A child by field id

Usage

```
node_child_by_field_id(x, id)
```

Arguments

x	A tree_sitter_node.
id	A field id (see language_field_id_for_name).

Value

A tree_sitter_node, or NULL.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  expr <- node_child(tree_root_node(text_parse("x <- 1", lang)), 1)
  id <- language_field_id_for_name(lang, "lhs")
  node_text(node_child_by_field_id(expr, id))
}
```

node_child_by_field_name *A child by field name*

Description

A child by field name

Usage

```
node_child_by_field_name(x, name)
```

Arguments

x	A tree_sitter_node.
name	A field name.

Value

A tree_sitter_node, or NULL.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  node_text(node_child_by_field_name(expr, "lhs"))
}
```

---

node_descendant_count *The number of descendants of a node*

Description

The number of descendants of a node

Usage

```
node_descendant_count(x)
```

Arguments

x	A tree_sitter_node.

Value

A single double, including the node itself.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_descendant_count(node)
}
```

node_grammar_type *The grammar type of a node*

Description

Like node_type, but returns the type as written in the grammar, ignoring aliases.

Usage

```
node_grammar_type(x)
```

Arguments

x	A tree_sitter_node.

Value

A single string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_grammar_type(node)
}
```

node_is_named *Is a node named?*

Description

Is a node named?

Usage

```
node_is_named(x)
```

Arguments

x	A tree_sitter_node.

Value

TRUE or FALSE.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  expr <- node_child(tree_root_node(tree), 1)
  vapply(node_children(expr), node_is_named, logical(1))
}
```

---

node_language *The language of a node*

Description

The language of a node

Usage

```
node_language(x)
```

Arguments

x	A tree_sitter_node.

Value

A tree_sitter_language.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  language_name(node_language(node))
}
```

---

node_range *A node's range*

Description

A node's range

Usage

```
node_range(x)
```

Arguments

x	A tree_sitter_node.

Value

A tree_sitter_range.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_range(node)
}
```

---

node_raw_s_expression *The raw s-expression of a node*

Description

The raw s-expression of a node

Usage

```
node_raw_s_expression(x)
```

Arguments

x	A tree_sitter_node.

Value

A single string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_raw_s_expression(node)
}
```

node_show_s_expression *Show a node's s-expression*

Description

Show a node's s-expression

Usage

```
node_show_s_expression(x)
```

Arguments

x	A tree_sitter_node.

Value

x, invisibly, called for its side effect of printing.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- 1", treesitter.r::language()))
  node_show_s_expression(node)
}
```

node_text *The text underlying a node*

Description

The text underlying a node

Usage

```
node_text(x)
```

Arguments

x	A tree_sitter_node.

Value

A single string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x <- f(1)", treesitter.r::language()))
  node_text(node_child(node, 1))
}
```

---

node_type *The type of a node*

Description

The type of a node

Usage

```
node_type(x)
```

Arguments

x	A tree_sitter_node.

Value

A single string.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  node <- tree_root_node(text_parse("x + 1", treesitter.r::language()))
  node_type(node)
}
```

---

parser *Create a parser*

Description

parser() constructs a parser from a tree_sitter_language. Use parser_parse to parse text with it.

Usage

```
parser(language)
```

Arguments

language	A tree_sitter_language, e.g. from a grammar package like treesitter.r::language().

Value

A tree_sitter_parser.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  p <- parser(treesitter.r::language())
  tree <- parser_parse(p, "x <- f(1, 2)")
  node_type(tree_root_node(tree))
}
```

---

parser-adjust *Adjust a parser*

Description

Each returns a new parser with one setting changed. - parser_set_language() sets the language. - parser_set_timeout() sets a parse timeout in microseconds (0 clears it). - parser_set_included_ranges() restricts parsing to a list of ranges (an empty list clears the restriction).

Usage

```
parser_set_language(x, language)

parser_set_timeout(x, timeout)

parser_set_included_ranges(x, included_ranges)
```

Arguments

x	A tree_sitter_parser.
language	A tree_sitter_language.
timeout	A single whole number of microseconds.
included_ranges	A list of tree_sitter_range objects.

Value

A new tree_sitter_parser.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  p <- parser_set_timeout(parser(lang), 1e6)
  p <- parser_set_language(p, lang)
  first_line <- range(0, point(0, 0), 6, point(0, 6))
  p <- parser_set_included_ranges(p, list(first_line))
  node_text(tree_root_node(parser_parse(p, "x <- 1\ny <- 2")))
}
```

`parser-parse` *Parse or reparse text*

Description

- `parser_parse()` parses `text` and returns a `tree_sitter_tree`. - `parser_reparse()` performs an incremental reparse of a slightly edited `text`, reusing the old `tree`. All bytes and points are 0-indexed.

Usage

```
parser_parse(x, text, ...)

parser_reparse(
  x,
  text,
  tree,
  start_byte,
  start_point,
  old_end_byte,
  old_end_point,
  new_end_byte,
  new_end_point
)
```

Arguments

`x`	A `tree_sitter_parser`.
`text`	A single string to parse.
`...`	Unused.
`tree`	The original `tree_sitter_tree` from `parser_parse()`.
`start_byte`, `old_end_byte`, `new_end_byte`	Edit byte offsets.
`start_point`, `old_end_point`, `new_end_point`	Edit `tree_sitter_point`s.

Value

A tree_sitter_tree.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  p <- parser(treesitter.r::language())
  tree <- parser_parse(p, "x <- 1")
  # Replace the "1" at byte 5 with "42", then reparse incrementally.
  tree2 <- parser_reparse(p, "x <- 42", tree, 5, point(0, 5), 6, point(0, 6),
                          7, point(0, 7))
  node_text(tree_root_node(tree2))
}
```

---

points *Points*

Description

- point() creates a tree-sitter point. Points are 0-indexed. - point_row() and point_column() access the row and column. - is_point() tests for a point.

Usage

```
point(row, column)

point_row(x)

point_column(x)

is_point(x)
```

Arguments

x	A tree_sitter_point.
row, column	A 0-indexed row / column (single whole number).

Value

point() a point; the accessors a double; is_point() a logical.

Examples

```
p <- point(2, 4)
p
point_row(p)
point_column(p)
is_point(p)
```

query *Compile a tree-sitter query*

Description

Compile a tree-sitter query

Usage

```
query(language, source)
```

Arguments

language	A tree_sitter_language.
source	A single string of query source (tree-sitter S-expression pattern syntax).

Value

A tree_sitter_query.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  q <- query(treesitter.r::language(), "(call function: (identifier) @fn)")
  query_capture_count(q)
}
```

query-bytes *Query byte ranges per pattern (1-indexed pattern)*

Description

Query byte ranges per pattern (1-indexed pattern)

Usage

```
query_start_byte_for_pattern(x, i)

query_end_byte_for_pattern(x, i)
```

Arguments

x	A tree_sitter_query.
i	A 1-indexed pattern position.

Value

A single double (0-indexed byte).

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  q <- query(treesitter.r::language(), "(call) @call (identifier) @id")
  c(query_start_byte_for_pattern(q, 2), query_end_byte_for_pattern(q, 2))
}
```

---

query-counts *Query counts*

Description

Query counts

Usage

```
query_pattern_count(x)

query_capture_count(x)

query_string_count(x)
```

Arguments

x	A tree_sitter_query.

Value

A single double.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  q <- query(treesitter.r::language(), "(call function: (identifier) @fn)")
  c(query_pattern_count(q), query_capture_count(q), query_string_count(q))
}
```

query_captures *Run a query and collect its captures*

Description

Run a query and collect its captures

Usage

```
query_captures(x, node)
```

Arguments

x	A tree_sitter_query.
node	A tree_sitter_node to search within.

Value

A list with name (character vector of capture names) and node (list of tree_sitter_nodes), one entry per capture in traversal order.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  node <- tree_root_node(text_parse("f(1); g(2)", lang))
  q <- query(lang, "(call function: (identifier) @fn)")
  vapply(query_captures(q, node)$node, node_text, character(1))
}
```

query_matches *Run a query and collect its matches*

Description

Unlike treesitter, which nests matches by pattern, this returns a flat list of matches. Each match is list(pattern, name, node): the 0-indexed pattern, the capture names, and the captured tree_sitter_nodes.

Usage

```
query_matches(x, node)
```

Arguments

x	A tree_sitter_query.
node	A tree_sitter_node to search within.

Value

A list of matches.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  lang <- treesitter.r::language()
  node <- tree_root_node(text_parse("f(1); g(2)", lang))
  q <- query(lang, "(call function: (identifier) @fn)")
  matches <- query_matches(q, node)
  vapply(matches, function(m) node_text(m$node[[1]]), character(1))
}
```

---

ranges *Ranges*

Description

- `range()` creates a tree-sitter range from a start/end byte and point. - `range_start_byte()`, `range_start_point()`, `range_end_byte()`, and `range_end_point()` access the components. - `is_range()` tests for a range.

All bytes and points are 0-indexed.

Usage

```
range(start_byte, start_point, end_byte, end_point)

range_start_byte(x)

range_start_point(x)

range_end_byte(x)

range_end_point(x)

is_range(x)
```

Arguments

`x`	A `tree_sitter_range`.
`start_byte`, `end_byte`	Single whole numbers.
`start_point`, `end_point`	`tree_sitter_point` objects.

Value

`range()` a range; accessors their component; `is_range()` a logical.

Examples

```
r <- range(0, point(0, 0), 6, point(0, 6))
r
range_end_byte(r)
range_end_point(r)
is_range(r)
```

---

text_parse *Parse text in one step*

Description

text_parse() is a convenience wrapper that builds a parser for language and parses x with it, returning the tree directly.

Usage

```
text_parse(x, language)
```

Arguments

x	A single string to parse.
language	A tree_sitter_language.

Value

A tree_sitter_tree.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("f <- function(x) x + 1", treesitter.r::language())
  node_type(tree_root_node(tree))
}
```

---

tree-accessors *Tree accessors*

Description

- tree_text() returns the text the tree was parsed with. - tree_language() returns the tree's tree_sitter_language.

Usage

```
tree_text(x)

tree_language(x)
```

Arguments

x	A tree_sitter_tree.

Value

tree_text() a string; tree_language() a language.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  tree_text(tree)
  language_name(tree_language(tree))
}
```

---

tree-cursor *Walk a tree or node with a cursor*

Description

tree_walk() and node_walk() create a tree_sitter_tree_cursor, a mutable cursor for efficient traversal. Navigate with its methods: cursor$goto_first_child(), cursor$goto_next_sibling(), cursor$goto_parent() (each returns TRUE/FALSE), and read state with cursor$node(), cursor$field_name(), and cursor$depth().

Usage

```
tree_walk(x)

node_walk(node)
```

Arguments

x	A tree_sitter_tree.
node	A tree_sitter_node.

Value

A tree_sitter_tree_cursor.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  cursor <- tree_walk(text_parse("x <- 1", treesitter.r::language()))
  cursor$goto_first_child()
  node_type(cursor$node())
}
```

---

tree_included_ranges *The tree's included ranges*

Description

The tree's included ranges

Usage

```
tree_included_ranges(x)
```

Arguments

x	A tree_sitter_tree.

Value

A list of tree_sitter_range objects.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  tree_included_ranges(tree)
}
```

---

tree_root_node *Retrieve the root node of a tree*

Description

Retrieve the root node of a tree

Usage

```
tree_root_node(x)
```

Arguments

x	A tree_sitter_tree.

Value

A tree_sitter_node.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  node_type(tree_root_node(tree))
}
```

---

tree_root_node_with_offset

An offset root node

Description

Returns the root node shifted forward by byte and point, so positions read in the coordinate space of a larger document.

Usage

```
tree_root_node_with_offset(x, byte, point)
```

Arguments

x	A tree_sitter_tree.
byte	A 0-indexed byte offset.
point	A tree_sitter_point offset.

Value

A tree_sitter_node.

Examples

```
if (requireNamespace("treesitter.r", quietly = TRUE)) {
  tree <- text_parse("x <- 1", treesitter.r::language())
  node <- tree_root_node_with_offset(tree, 10, point(2, 0))
  node_start_point(node)
}
```

Index