Package 'treesitter'

June 6, 2025

Title Bindings to 'Tree-Sitter'

Version 0.3.0

Description Provides bindings to 'Tree-sitter', an incremental parsing system for programming tools. 'Tree-sitter' builds concrete syntax trees for source files of any language, and can efficiently update those syntax trees as the source file is edited. It also includes a robust error recovery system that provides useful parse results even in the presence of syntax errors.

License MIT + file LICENSE

URL https://github.com/DavisVaughan/r-tree-sitter, https://davisvaughan.github.io/r-tree-sitter/

BugReports https://github.com/DavisVaughan/r-tree-sitter/issues

Depends R (>= 4.3.0)

Imports cli (>= 3.6.2), R6 (>= 2.5.1), rlang (>= 1.1.3), vctrs (>= 0.6.5)

Suggests testthat (>= 3.0.0), treesitter.r (>= 1.1.0)

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation yes

Author Davis Vaughan [aut, cre],
Posit Software, PBC [cph, fnd],
Tree-sitter authors [cph] (Tree-sitter C library)

Maintainer Davis Vaughan <davis@posit.co>

Repository CRAN

Date/Publication 2025-06-06 15:50:01 UTC

Contents

`is_language` *Is* `x` *a language?*

Description

Use `is_language()` to determine if an object has a class of `"tree_sitter_language"`.

Usage

```
is_language(x)
```

Arguments

`x`	`[object]` An object.

Value

- `TRUE` if `x` is a `"tree_sitter_language"`.
- `FALSE` otherwise.

Examples

```
language <- treesitter.r::language()
is_language(language)
```

`is_node` *Is* `x` *a node?*

Description

Checks if `x` is a `tree_sitter_node` or not.

Usage

```
is_node(x)
```

Arguments

`x`	`[object]` An object.

Value

TRUE if x is a tree_sitter_node, otherwise FALSE.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

is_node(node)

is_node(1)
```

---

is_parser *Is* x *a parser?*

Description

Checks if x is a tree_sitter_parser or not.

Usage

```
is_parser(x)
```

Arguments

x	[object] An object.

Value

TRUE if x is a tree_sitter_parser, otherwise FALSE.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

is_parser(parser)

is_parser(1)
```

is_query *Is* x *a query?*

Description

Checks if x is a tree_sitter_query or not.

Usage

```
is_query(x)
```

Arguments

x	[object] An object.

Value

TRUE if x is a tree_sitter_query, otherwise FALSE.

Examples

```
source <- "(identifier) @id"
language <- treesitter.r::language()

query <- query(language, source)

is_query(query)

is_query(1)
```

is_tree *Is* x *a tree?*

Description

Checks if x is a tree_sitter_tree or not.

Usage

```
is_tree(x)
```

Arguments

x	[object] An object.

Value

TRUE if x is a tree_sitter_tree, otherwise FALSE.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)

is_tree(tree)

is_tree(1)
```

---

language_field_count *Language field count*

Description

Get the number of fields contained within a language.

Usage

```
language_field_count(x)
```

Arguments

x	[tree_sitter_language] A tree-sitter language object.

Value

A single double value.

Examples

```
language <- treesitter.r::language()
language_field_count(language)
```

`language_field_id_for_name` *Language field identifiers*

Description

Get the integer field identifier for a field name. If you are going to be using a field name repeatedly, it is often a little faster to use the corresponding field identifier instead.

Usage

```
language_field_id_for_name(x, name)
```

Arguments

`x`	`[tree_sitter_language]` A tree-sitter language object.
`name`	`[character]` The language field names to look up field identifiers for.

Value

An integer vector the same length as `name` containing:

- The field identifier for the field name, if known.
- `NA`, if the field name was not known.

See Also

`language_field_name_for_id()`

Examples

```
language <- treesitter.r::language()
language_field_id_for_name(language, "lhs")
```

language_field_name_for_id *Language field names*

Description

Get the field name for a field identifier.

Usage

```
language_field_name_for_id(x, id)
```

Arguments

x	[tree_sitter_language] A tree-sitter language object.
id	[integer] The language field identifiers to look up field names for.

Value

A character vector the same length as `id` containing:

- The field name for the field identifier, if known.
- `NA`, if the field identifier was not known.

See Also

`language_field_id_for_name()`

Examples

```
language <- treesitter.r::language()
language_field_name_for_id(language, 1)
```

language_name *Language name*

Description

Extract a language object's language name.

Usage

```
language_name(x)
```

Arguments

x	[tree_sitter_language] A tree-sitter language object.

Value

A string.

Examples

```
language <- treesitter.r::language()
language_name(language)
```

---

language_next_state *Language state advancement*

Description

Get the next state in the grammar.

Usage

```
language_next_state(x, state, symbol)
```

Arguments

x	[tree_sitter_language] A tree-sitter language object.
state, symbol	[integer] Vectors of equal length containing the current state and symbol information.

Value

A single integer representing the next state.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to function definition
node <- node_child(node, 1)
node <- node_child(node, 3)
```

```
node

state <- node_parse_state(node)
symbol <- node_grammar_symbol(node)

# Function definition symbol
language_symbol_name(language, 85)

# Next state (this is all grammar dependent)
language_next_state(language, state, symbol)
```

`language_state_count` *Language state count*

Description

Get the number of states traversable within a language.

Usage

```
language_state_count(x)
```

Arguments

`x`	`[tree_sitter_language]` A tree-sitter language object.

Value

A single double value.

Examples

```
language <- treesitter.r::language()
language_state_count(language)
```

language_symbol_count *Language symbol count*

Description

Get the number of symbols contained within a language.

Usage

```
language_symbol_count(x)
```

Arguments

x	[tree_sitter_language] A tree-sitter language object.

Value

A single double value.

Examples

```
language <- treesitter.r::language()
language_symbol_count(language)
```

language_symbol_for_name *Language symbols*

Description

Get the integer symbol ID for a particular node name. Can be useful for exploring the grammar.

Usage

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

Arguments

x	[tree_sitter_language] A tree-sitter language object.
name	[character] The names to look up symbols for.
...	These dots are for future extensions and must be empty.
named	[logical] Should named or anonymous nodes be looked up? Recycled to the size of name.

Value

An integer vector the same size as `name` containing either:

- The integer symbol ID of the node name, if known.
- `NA` if the node name was not known.

See Also

`language_symbol_name()`

Examples

```
language <- treesitter.r::language()
language_symbol_for_name(language, "identifier")
```

`language_symbol_name` *Language symbol names*

Description

Get the name for a particular language symbol ID. Can be useful for exploring a grammar.

Usage

```
language_symbol_name(x, symbol)
```

Arguments

`x`	`[tree_sitter_language]` A tree-sitter language object.
`symbol`	`[positive integer]` The language symbols to look up names for.

Value

A character vector the same length as `symbol` containing:

- The name of the symbol, if known.
- `NA`, if the symbol was not known.

See Also

`language_symbol_for_name()`

Examples

```
language <- treesitter.r::language()
language_symbol_name(language, 1)
```

node-child *Get a node's child by index*

Description

These functions return the ith child of x.

- node_child() considers both named and anonymous children.
- node_named_child() considers only named children.

Usage

```
node_child(x, i)

node_named_child(x, i)
```

Arguments

x	[tree_sitter_node] A node.
i	[integer(1)] The index of the child to return.

Value

The ith child node of x or NULL if there is no child at that index.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Starts with `program` node for the whole document
node

# Navigate to first child
node <- node_child(node, 1)
node

# Note how the named variant skips the anonymous operator node
node_child(node, 2)
node_named_child(node, 2)

# OOB indices return `NULL`
node_child(node, 5)
```

node-child-by-field *Get a node's child by field id or name*

Description

These functions return children of `x` by field id or name.

- `node_child_by_field_id()` retrieves a child by field id.
- `node_child_by_field_name()` retrieves a child by field name.

Use `language_field_id_for_name()` to get the field id for a field name.

Usage

```
node_child_by_field_id(x, id)

node_child_by_field_name(x, name)
```

Arguments

`x`	`[tree_sitter_node]` A node.
`id`	`[integer(1)]` The field id of the child to return.
`name`	`[character(1)]` The field name of the child to return.

Value

A child of `x`, or `NULL` if no matching child can be found.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)
node

# Get the field name of the first child
name <- node_field_name_for_child(node, 1)
name
```

```
# Now get the child again by that field name
node_child_by_field_name(node, name)

# If you need to look up by field name many times, you can look up the
# more direct field id first and use that instead
id <- language_field_id_for_name(language, name)
id

node_child_by_field_id(node, id)

# Returns `NULL` if no matching child
node_child_by_field_id(node, 10000)
```

`node-child-count` *Get a node's child count*

Description

These functions return the number of children of `x`.

- `node_child_count()` considers both named and anonymous children.
- `node_named_child_count()` considers only named children.

Usage

```
node_child_count(x)

node_named_child_count(x)
```

Arguments

`x`	`[tree_sitter_node]` A node.

Value

A single integer, the number of children of `x`.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)
```

```
node

# Note how the named variant doesn't count the anonymous operator node
node_child_count(node)
node_named_child_count(node)
```

`node-children` *Get a node's children*

Description

These functions return the children of `x` within a list.

- `node_children()` considers both named and anonymous children.
- `node_named_children()` considers only named children.

Usage

```
node_children(x)

node_named_children(x)
```

Arguments

`x`	`[tree_sitter_node]` A node.

Value

The children of `x` as a list.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)
node

# Note how the named variant doesn't include the anonymous operator node
node_children(node)
node_named_children(node)
```

node-descendant *Node descendants*

Description

These functions return the smallest node within this node that spans the given range of bytes or points. If the ranges are out of bounds, or no smaller node can be determined, the input is returned.

Usage

```
node_descendant_for_byte_range(x, start, end)

node_named_descendant_for_byte_range(x, start, end)

node_descendant_for_point_range(x, start, end)

node_named_descendant_for_point_range(x, start, end)
```

Arguments

x	[tree_sitter_node] A node.
start, end	[integer(1) / tree_sitter_point] For the byte range functions, start and end bytes to search within. For the point range functions, start and end points created by `point()` to search within.

Value

A node.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# The whole `<-` binary operator node
node <- node_child(node, 1)
node

# The byte range points to a location in the word `function`
node_descendant_for_byte_range(node, 7, 9)
node_named_descendant_for_byte_range(node, 7, 9)
```

```
start <- point(0, 14)
end <- point(0, 15)

node_descendant_for_point_range(node, start, end)
node_named_descendant_for_point_range(node, start, end)

# OOB returns the input
node_descendant_for_byte_range(node, 25, 29)
```

`node-field-name-for-child`

Get a child's field name by index

Description

These functions return the field name for the `i`th child of `x`.

- `node_field_name_for_child()` considers both named and anonymous children.
- `node_field_name_for_named_child()` considers only named children.

Nodes themselves don't know their own field names, because they don't know if they are fields or not. You must have access to their parents to query their field names.

Usage

```
node_field_name_for_child(x, i)

node_field_name_for_named_child(x, i)
```

Arguments

`x`	`[tree_sitter_node]` A node.
`i`	`[integer(1)]` The index of the child to get the field name for.

Value

The field name for the `i`th child of `x`, or `NA_character_` if that child doesn't exist.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)
```

```
# Navigate to first child
node <- node_child(node, 1)
node

# Get the field name of the first few children (note that anonymous children
# are considered)
node_field_name_for_child(node, 1)
node_field_name_for_child(node, 2)

# Get the field name of the first few named children (note that anonymous
# children are not considered)
node_field_name_for_named_child(node, 1)
node_field_name_for_named_child(node, 2)

# 10th child doesn't exist, this returns `NA_character_`
node_field_name_for_child(node, 10)
```

`node-first-child-byte` *Get the first child that extends beyond the given byte offset*

Description

These functions return the first child of `x` that extends beyond the given `byte` offset. Note that `byte` is a 0-indexed offset.

- `node_first_child_for_byte()` considers both named and anonymous nodes.
- `node_first_named_child_for_byte()` considers only named nodes.

Usage

```
node_first_child_for_byte(x, byte)

node_first_named_child_for_byte(x, byte)
```

Arguments

`x`	`[tree_sitter_node]` A node.
`byte`	`[integer(1)]` The byte to start the search from. Note that `byte` is 0-indexed!

Value

A new node, or `NULL` if there is no node past the `byte` offset.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)
node

# `fn {here}<- function()`
node_first_child_for_byte(node, 3)
node_first_named_child_for_byte(node, 3)

# Past any node
node_first_child_for_byte(node, 100)
```

---

node-grammar *Node grammar types and symbols*

Description

- `node_grammar_type()` gets the node's type as it appears in the grammar, *ignoring aliases*.
- `node_grammar_symbol()` gets the node's symbol (the type as a numeric id) as it appears in the grammar, *ignoring aliases*. This should be used in `language_next_state()` rather than `node_symbol()`.

Usage

```
node_grammar_type(x)

node_grammar_symbol(x)
```

Arguments

x	`[tree_sitter_node]` A node.

See Also

`node_type()`, `node_symbol()`

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Examples for these functions are highly specific to the grammar,
# because they relies on the placement of `alias()` calls in the grammar.
node_grammar_type(node)
node_grammar_symbol(node)
```

---

node-location *Node byte and point accessors*

Description

These functions return information about the location of `x` in the document. The byte, row, and column locations are all 0-indexed.

- `node_start_byte()` returns the start byte.
- `node_end_byte()` returns the end byte.
- `node_start_point()` returns the start point, containing a row and column location within the document. Use accessors like `point_row()` to extract the row and column positions.
- `node_end_point()` returns the end point, containing a row and column location within the document. Use accessors like `point_row()` to extract the row and column positions.
- `node_range()` returns a range object that contains all of the above information. Use accessors like `range_start_point()` to extract individual pieces from the range.

Usage

```
node_start_byte(x)

node_end_byte(x)

node_start_point(x)

node_end_point(x)

node_range(x)
```

Arguments

`x`	`[tree_sitter_node]` A node.

Value

- `node_start_byte()` and `node_end_byte()` return a single numeric value.
- `node_start_point()` and `node_end_point()` return single points.
- `node_range()` returns a range.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)

# Navigate to function definition node
node <- node_child(node, 3)
node

node_start_byte(node)
node_end_byte(node)

node_start_point(node)
node_end_point(node)

node_range(node)
```

`node-metadata` *Node metadata*

Description

These functions return metadata about the current node.

- `node_is_named()` reports if the current node is named or anonymous.
- `node_is_missing()` reports if the current node is `MISSING`, i.e. if it was implied through error recovery.
- `node_is_extra()` reports if the current node is an "extra" from the grammar.
- `node_is_error()` reports if the current node is an `ERROR` node.
- `node_has_error()` reports if the current node is an `ERROR` node, or if any descendants of the current node are `ERROR` or `MISSING` nodes.

Usage

```
node_is_named(x)

node_is_missing(x)

node_is_extra(x)

node_is_error(x)

node_has_error(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

TRUE or FALSE.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node <- node_child(node, 1)

fn <- node_child(node, 1)
operator <- node_child(node, 2)

fn
node_is_named(fn)

operator
node_is_named(operator)

# Examples of `TRUE` cases for these are a bit hard to come up with, because
# they are dependent on the exact state of the grammar and the error recovery
# algorithm
node_is_missing(node)
node_is_extra(node)
```

node-parse-state *Node parse states*

Description

These are advanced functions that return information about the internal parse states.

- `node_parse_state()` returns the parse state of the current node.
- `node_next_parse_state()` returns the parse state after this node.

See `language_next_state()` for more information.

Usage

```
node_parse_state(x)

node_next_parse_state(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A single integer representing a parse state.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node <- node_child(node, 1)

# Parse states are grammar dependent
node_parse_state(node)
node_next_parse_state(node)
```

node-sibling *Node sibling accessors*

Description

These functions return siblings of the current node, i.e. if you looked "left" or "right" from the current node rather "up" (parent) or "down" (child).

- `node_next_sibling()` and `node_next_named_sibling()` return the next sibling.
- `node_previous_sibling()` and `node_previous_named_sibling()` return the previous sibling.

Usage

```
node_next_sibling(x)

node_next_named_sibling(x)

node_previous_sibling(x)

node_previous_named_sibling(x)
```

Arguments

x	`[tree_sitter_node]` A node.

Value

A sibling node, or `NULL` if there is no sibling node.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Navigate to first child
node <- node_child(node, 1)

# Navigate to function definition node
node <- node_child(node, 3)
node

node_previous_sibling(node)
```

```
# Skip anonymous operator node
node_previous_named_sibling(node)

# There isn't one!
node_next_sibling(node)
```

`node_descendant_count` *Node descendant count*

Description

Returns the number of descendants of this node, including this node in the count.

Usage

```
node_descendant_count(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A single double.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Top level program node
node_descendant_count(node)

# The whole `<-` binary operator node
node <- node_child(node, 1)
node_descendant_count(node)

# Just the literal `<-` operator itself
node <- node_child_by_field_name(node, "operator")
node_descendant_count(node)
```

node_language *Get a node's underlying language*

Description

node_language() returns the document text underlying a node.

Usage

```
node_language(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A tree_sitter_language object.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node_language(node)
```

node_parent *Get a node's parent*

Description

node_parent() looks up the tree and returns the current node's parent.

Usage

```
node_parent(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

The parent node of `x` or `NULL` if there is no parent.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Parent of a root node is `NULL`
node_parent(node)

node_function <- node |>
  node_child(1) |>
  node_child(3)

node_function

node_parent(node_function)
```

`node_raw_s_expression` *"Raw" S-expression*

Description

`node_raw_s_expression()` returns the "raw" s-expression as seen by tree-sitter. Most of the time, `node_show_s_expression()` provides a better view of the tree, but occasionally it can be useful to see exactly what the underlying C library is using.

Usage

```
node_raw_s_expression(x)
```

Arguments

`x`	`[tree_sitter_node]` A node.

Value

A single string containing the raw s-expression.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node_raw_s_expression(node)
```

`node_show_s_expression` *Pretty print a* node*'s s-expression*

Description

`node_show_s_expression()` prints a nicely formatted s-expression to the console. It powers the print methods of nodes and trees.

Usage

```
node_show_s_expression(
  x,
  ...,
  max_lines = NULL,
  show_anonymous = TRUE,
  show_locations = TRUE,
  show_parentheses = TRUE,
  dangling_parenthesis = TRUE,
  color_parentheses = TRUE,
  color_locations = TRUE
)
```

Arguments

`x`	`[tree_sitter_node]` A node.
`...`	These dots are for future extensions and must be empty.
`max_lines`	`[double(1) / NULL]` An optional maximum number of lines to print. If the maximum is hit, then `<truncated>` will be printed at the end.
`show_anonymous`	`[bool]` Should anonymous nodes be shown? If `FALSE`, only named nodes are shown.
`show_locations`	`[bool]` Should node locations be shown?

show_parentheses
: [bool]
Should parentheses around each node be shown?

dangling_parenthesis
: [bool]
Should the) parenthesis "dangle" on its own line? If FALSE, it is appended to the line containing the last child. This can be useful for conserving space.

color_parentheses
: [bool]
Should parentheses be colored? Printing large s-expressions is faster if this is set to FALSE.

color_locations
: [bool]
Should locations be colored? Printing large s-expressions is faster if this is set to FALSE.

Value

x invisibly.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function(a, b = 2) { a + b + 2 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node_show_s_expression(node)

node_show_s_expression(node, max_lines = 5)

# This is more like a typical abstract syntax tree
node_show_s_expression(
  node,
  show_anonymous = FALSE,
  show_locations = FALSE,
  dangling_parenthesis = FALSE
)
```

---

node_symbol *Node symbol*

Description

node_symbol() returns the symbol id of the current node as an integer.

Usage

```
node_symbol(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A single integer.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Top level program node
node_symbol(node)

# The whole `<-` binary operator node
node <- node_child(node, 1)
node_symbol(node)

# Just the literal `<-` operator itself
node <- node_child_by_field_name(node, "operator")
node_symbol(node)
```

---

node_text *Get a node's underlying text*

Description

`node_text()` returns the document text underlying a node.

Usage

```
node_text(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A single string containing the node's text.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

node |>
  node_child(1) |>
  node_child_by_field_name("rhs") |>
  node_text()
```

---

node_type *Node type*

Description

`node_type()` returns the "type" of the current node as a string.

This is a very useful function for making decisions about how to handle the current node.

Usage

```
node_type(x)
```

Arguments

x	`[tree_sitter_node]` A node.

Value

A single string.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Top level program node
```

```
node_type(node)

# The whole `<-` binary operator node
node <- node_child(node, 1)
node
node_type(node)

# Just the literal `<-` operator itself
node <- node_child_by_field_name(node, "operator")
node
node_type(node)
```

---

node_walk *Generate a* TreeCursor *iterator*

Description

node_walk() creates a TreeCursor starting at the current node. You can use it to "walk" the tree more efficiently than using node_child() and other similar node functions.

Usage

```
node_walk(x)
```

Arguments

x	[tree_sitter_node] A node.

Value

A TreeCursor object.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

cursor <- node_walk(node)

cursor$goto_first_child()
cursor$goto_first_child()
cursor$node()
cursor$goto_next_sibling()
cursor$node()
```

parser *Create a new parser*

Description

parser() constructs a parser from a tree-sitter language object. You can use parser_parse() to parse language specific text with it.

Usage

```
parser(language)
```

Arguments

language	[tree_sitter_language] A language object.

Value

A new parser.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)
parser

text <- "1 + foo"
tree <- parser_parse(parser, text)
tree
```

parser-adjustments *Parser adjustments*

Description

- parser_set_language() sets the language of the parser. This is usually done by parser() though.
- parser_set_timeout() sets an optional timeout used when calling parser_parse() or parser_reparse(). If the timeout is hit, an error occurs.
- parser_set_included_ranges() sets an optional list of ranges that are the only locations considered when parsing. The ranges are created by range().

Usage

```
parser_set_language(x, language)

parser_set_timeout(x, timeout)

parser_set_included_ranges(x, included_ranges)
```

Arguments

x	[tree_sitter_parser] A parser.
language	[tree_sitter_language] A language.
timeout	[double(1)] A single whole number corresponding to a timeout in microseconds to use when parsing.
included_ranges	[list_of<tree_sitter_range>] A list of ranges constructed by range(). These are the only locations that will be considered when parsing. An empty list can be used to clear any existing ranges so that the parser will again parse the entire document.

Value

A new parser.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)
parser_set_timeout(parser, 10000)
```

---

parser-parse *Parse or reparse text*

Description

- parser_parse() performs an initial parse of text, a string typically containing contents of a file. It returns a tree for further manipulations.
- parser_reparse() performs a fast incremental reparse. text is typically a slightly modified version of the original text with a new "edit" applied. The position of the edit is described by the byte and point arguments to this function. The tree argument corresponds to the original tree returned by parser_parse().

All bytes and points should be 0-indexed.

Usage

```
parser_parse(x, text, ..., encoding = "UTF-8")

parser_reparse(
  x,
  text,
  tree,
  start_byte,
  start_point,
  old_end_byte,
  old_end_point,
  new_end_byte,
  new_end_point,
  ...,
  encoding = "UTF-8"
)
```

Arguments

`x`	`[tree_sitter_parser]` A parser.
`text`	`[string]` The text to parse.
`...`	These dots are for future extensions and must be empty.
`encoding`	`[string]` The expected encoding of the `text`. Either `"UTF-8"` or `"UTF-16"`.
`tree`	`[tree_sitter_tree]` The original tree returned by `parser_parse()`. Components of the tree will be reused to perform the incremental reparse.
`start_byte`, `start_point`	`[double(1) / tree_sitter_point]` The starting byte and starting point of the edit location.
`old_end_byte`, `old_end_point`	`[double(1) / tree_sitter_point]` The old ending byte and old ending point of the edit location.
`new_end_byte`, `new_end_point`	`[double(1) / tree_sitter_point]` The new ending byte and new ending point of the edit location.

Value

A new `tree`.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)
```

```
text <- "1 + foo"
tree <- parser_parse(parser, text)
tree

text <- "1 + bar(foo)"
parser_reparse(
  parser,
  text,
  tree,
  start_byte = 4,
  start_point = point(0, 4),
  old_end_byte = 7,
  old_end_point = point(0, 7),
  new_end_byte = 12,
  new_end_point = point(0, 12)
)
```

`points` *Points*

Description

- `point()` creates a new tree-sitter point.
- `point_row()` and `point_column()` access a point's row and column value, respectively.
- `is_point()` determines whether or not an object is a point.

Note that points are 0-indexed. This is typically the easiest form to work with them in, since most of the time when you are provided row and column information from third party libraries, they will already be 0-indexed. It is also consistent with bytes, which are also 0-indexed and are often provided alongside their corresponding points.

Usage

```
point(row, column)

point_row(x)

point_column(x)

is_point(x)
```

Arguments

`row`	`[double(1)]` A 0-indexed row to place the point at.
`column`	`[double(1)]` A 0-indexed column to place the point at.

x [tree_sitter_point]
A point.

Value

- `point()` returns a new point.
- `point_row()` and `point_column()` return a single double.
- `is_point()` returns `TRUE` or `FALSE`.

Examples

```
x <- point(1, 2)

point_row(x)
point_column(x)

is_point(x)
```

---

query *Queries*

Description

`query()` lets you specify a query `source` string for use with `query_captures()` and `query_matches()`. The `source` string is written in a way that is somewhat similar to the idea of capture groups in regular expressions. You write out one or more query patterns that match nodes in a tree, and then you "capture" parts of those patterns with `@name` tags. The captures are the values returned by `query_captures()` and `query_matches()`. There are also a series of *predicates* that can be used to further refine the query. Those are described in the `query_matches()` help page.

Read the tree-sitter documentation to learn more about the query syntax.

Usage

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

Arguments

language [tree_sitter_language]
A language.

source [string]
A query source string.

Value

A query.

Storing queries

Query objects contain *external pointers*, so they cannot be saved to disk and reloaded. One consequence of this is you cannot create them at build time inside your package. For example, to precompile a query you may assume you can create a global variable in your package with top level code like this:

```
QUERY <- treesitter::query(treesitter.r::language(), "query_source_text")
```

This won't work for two reasons:

- The external query in `QUERY` is created at package build time, and is no longer valid at package load time.
- The version of treesitter and treesitter.r are locked to the version used at build time, rather than at package load time.

The correct way to do this is to create the query on package load, like this:

```
QUERY <- NULL

.onLoad <- function(libname, pkgname) {
  QUERY <<- treesitter::query(treesitter.r::language(), "query_source_text")
}
```

This is one place where usage of `<<-` is acceptable.

Examples

```
# This query looks for binary operators where the left hand side is an
# identifier named `fn`, and the right hand side is a function definition.
# The operator can be `<-` or `=` (technically it can also be things like
# `+` as well in this example).
source <- '(binary_operator
  lhs: (identifier) @lhs
  operator: _ @operator
  rhs: (function_definition) @rhs
  (#eq? @lhs "fn")
)'

language <- treesitter.r::language()

query <- query(language, source)

text <- "
  fn <- function() {}
  fn2 <- function() {}
  fn <- 5
  fn = function(a, b, c) { a + b + c }
"
parser <- parser(language)
tree <- parser_parse(parser, text)
```

```
node <- tree_root_node(tree)

query_matches(query, node)
```

---

query-accessors *Query accessors*

Description

- `query_pattern_count()` returns the number of patterns in a query.
- `query_capture_count()` returns the number of captures in a query.
- `query_string_count()` returns the number of string literals in a query.
- `query_start_byte_for_pattern()` and `query_end_byte_for_pattern()` return the byte where the `i`th pattern starts/ends in the query `source`.

Usage

```
query_pattern_count(x)

query_capture_count(x)

query_string_count(x)

query_start_byte_for_pattern(x, i)

query_end_byte_for_pattern(x, i)
```

Arguments

`x`	`[tree_sitter_query]` A query.
`i`	`[double(1)]` The `i`th pattern to extract the byte for.

Value

- `query_pattern_count()`, `query_capture_count()`, and `query_string_count()` return a single double count value.
- `query_start_byte_for_pattern()` and `query_end_byte_for_pattern()` return a single double for their respective byte if there was an `i`th pattern, otherwise they return `NA`.

Examples

```
source <- '(binary_operator
  lhs: (identifier) @lhs
  operator: _ @operator
  rhs: (function_definition) @rhs
  (#eq? @lhs "fn")
)'
language <- treesitter.r::language()

query <- query(language, source)

query_pattern_count(query)
query_capture_count(query)
query_string_count(query)

query_start_byte_for_pattern(query, 1)
query_end_byte_for_pattern(query, 1)

text <- "
  fn <- function() {}
  fn2 <- function() {}
  fn <- 5
  fn <- function(a, b, c) { a + b + c }
"
parser <- parser(language)
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

query_matches(query, node)
```

---

query-matches-and-captures

Query matches and captures

Description

These two functions execute a query on a given node, and return the captures of the query for further use. Both functions return the same information, just structured differently depending on your use case.

- `query_matches()` returns the captures first grouped by *pattern*, and further grouped by *match* within each pattern. This is useful if you include multiple patterns in your query.
- `query_captures()` returns a flat list of captures ordered by their node location in the original text. This is normally the easiest structure to use if you have a single pattern without any alternations that would benefit from having individual captures split by match.

Both also return the capture name, i.e. the `@name` you specified in your query.

Usage

```
query_matches(x, node, ..., range = NULL)

query_captures(x, node, ..., range = NULL)
```

Arguments

x	[tree_sitter_query] A query.
node	[tree_sitter_node] A node to run the query over.
...	These dots are for future extensions and must be empty.
range	[tree_sitter_range / NULL] An optional range to restrict the query to.

Predicates

There are 3 core types of predicates supported:

- `#eq? @capture "string"`
- `#eq? @capture1 @capture2`
- `#match? @capture "regex"`

Here are a few examples:

```
# Match an identifier named `"name-of-interest"`
(
  (identifier) @id
  (#eq? @id "name-of-interest")
)

# Match a binary operator where the left and right sides are the same name
(
  (binary_operator
    lhs: (identifier) @id1
    rhs: (identifier) @id2
  )
  (#eq? @id1 @id2)
)

# Match a name with a `_` in it
(
  (identifier) @id
  (#match? @id "_")
)
```

Each of these predicates can be inverted with a `not-` prefix.

```
(
  (identifier) @id
  (#not-eq? @id "name-of-interest")
)
```

Each of these predicates can be converted from an *all* style predicate to an *any* style predicate with an `any-` prefix. This is only useful with *quantified* captures, i.e. `(comment)+`, where the `+` specifies "one or more comment".

```
# Finds a block of comments where ALL comments are empty comments
(
  (comment)+ @comment
  (#eq? @comment "#")
)

# Finds a block of comments where ANY comments are empty comments
(
  (comment)+ @comment
  (#any-eq? @comment "#")
)
```

This is the full list of possible predicate permutations:

- `#eq?`
- `#not-eq?`
- `#any-eq?`
- `#any-not-eq?`
- `#match?`
- `#not-match?`
- `#any-match?`
- `#any-not-match?`

String double quotes:

The underlying tree-sitter predicate parser requires that strings supplied in a query must use double quotes, i.e. `"string"` not `'string'`. If you try and use single quotes, you will get a query error.

`#match?` **regex:**

The regex support provided by `#match?` is powered by `grepl()`.

Escapes are a little tricky to get right within these match regex strings. To use something like `\s` in the regex string, you need the literal text `\\s` to appear in the string to tell the tree-sitter regex engine to escape the backslash so you end up with just `\s` in the captured string. This requires putting two literal backslash characters in the R string itself, which can be accomplished with either `"\\\\s"` or using a raw string like `r'["\\\\s"]'` which is typically a little easier. You can also write your queries in a separate file (typically called `queries.scm`) and read them into R, which is also a little more straightforward because you can just write something like `(#match? @id "^\\s$")` and that will be read in correctly.

Examples

```
# ------------------------------------------------------------------------
# Simple query

text <- "
foo + b + a + ab
and(a)
"

source <- "
(identifier) @id
"

language <- treesitter.r::language()

query <- query(language, source)
parser <- parser(language)
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# A flat ordered list of captures, that's most useful here since
# we only have 1 pattern!
captures <- query_captures(query, node)
captures$node

# ------------------------------------------------------------------------
# Quantified query

text <- "
# this
# that
NULL

# and
# here
1 + 1

# there
2
"

# Find blocks of one or more comments
# The `+` is a regex `+` meaning "one or more" comments in a row
source <- "
(comment)+ @comment
"

language <- treesitter.r::language()

query <- query(language, source)
parser <- parser(language)
tree <- parser_parse(parser, text)
```

```
node <- tree_root_node(tree)

# The extra structure provided by `query_matches()` is useful here so
# we can see the 3 distinct blocks of comments
matches <- query_matches(query, node)

# We provided one query pattern, so lets extract that
matches <- matches[[1]]

# 3 blocks of comments
matches[[1]]
matches[[2]]
matches[[3]]

# ------------------------------------------------------------------------
# Multiple query patterns

# If you know you need to run multiple queries, you can run them all at once
# in one pass over the tree by providing multiple query patterns.

text <- "
a <- 1
b <- function() {}
c <- b
"

# Use an extra set of `()` to separate multiple query patterns
source <- "
(
  (identifier) @id
)
(
  (binary_operator) @binary
)
"

language <- treesitter.r::language()

query <- query(language, source)
parser <- parser(language)
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# The extra structure provided by `query_matches()` is useful here so
# we can separate the two queries
matches <- query_matches(query, node)

# First query - all identifiers
matches[[1]]

# Second query - all binary operators
matches[[2]]
```

```
# ------------------------------------------------------------------------
# The `#eq?` and `#match?` predicates

text <- '
fn(a, b)

test_that("this", {
  test
})

fn_name(args)

test_that("that", {
  test
})

fn2_(args)
'

language <- treesitter.r::language()
parser <- parser(language)
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Use an extra set of outer `()` when you are applying a predicate to ensure
# the query pattern is grouped with the query predicate.
# This one finds all function calls where the function name is `test_that`.
source <- '
(
  (call
    function: (identifier) @name
  ) @call
  (#eq? @name "test_that")
)
'

query <- query(language, source)

# It's fine to have a flat list of captures here, but we probably want to
# remove the `@name` captures and just retain the full `@call` captures.
captures <- query_captures(query, node)
captures$node[captures$name == "call"]

# This one finds all functions with a `_` in their name. It uses the R
# level `grepl()` for the regex processing.
source <- '
(
  (call
    function: (identifier) @name
  ) @call
  (#match? @name "_")
)
'
```

```
query <- query(language, source)

captures <- query_captures(query, node)
captures$node[captures$name == "call"]

# --------------------------------------------------------------------------
# The `any-` and `not-` predicate modifiers

text <- '
# 1
#
# 2
NULL

# 3
# 4
NULL

#
#
NULL

#
# 5
#
# 6
#
NULL
'

language <- treesitter.r::language()
parser <- parser(language)
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Two queries:
# - Find comment blocks where there is at least one empty comment
# - Find comment blocks where there is at least one non-empty comment
source <- '
(
  (comment)+ @comment
  (#any-eq? @comment "#")
)
(
  (comment)+ @comment
  (#any-not-eq? @comment "#")
)
'

query <- query(language, source)

matches <- query_matches(query, node)
```

```
# Query 1 has 3 comment blocks that match
query1 <- matches[[1]]
query1[[1]]
query1[[2]]
query1[[3]]

# Query 2 has 3 comment blocks that match (a different set than query 1!)
query2 <- matches[[2]]
query2[[1]]
query2[[2]]
query2[[3]]
```

---

ranges *Ranges*

Description

- `range()` creates a new tree-sitter range.
- `range_start_byte()` and `range_end_byte()` access a range's start and end bytes, respectively.
- `range_start_point()` and `range_end_point()` access a range's start and end points, respectively.
- `is_range()` determines whether or not an object is a range.

Note that the bytes and points used in ranges 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

`start_byte`, `end_byte`
: `[double(1)]`
0-indexed bytes for the start and end of the range, respectively.

start_point, end_point
: [tree_sitter_point]
0-indexed points for the start and end of the range, respectively.

x
: [tree_sitter_range]
A range.

Value

- `range()` returns a new range.
- `range_start_byte()` and `range_end_byte()` return a single double.
- `range_start_point()` and `range_end_point()` return a `point()`.
- `is_range()` returns `TRUE` or `FALSE`.

See Also

`node_range()`

Examples

```
x <- range(5, point(1, 3), 7, point(1, 5))
x

range_start_byte(x)
range_end_byte(x)

range_start_point(x)
range_end_point(x)

is_range(x)
```

---

text_parse *Parse a snippet of text*

Description

`text_parse()` is a convenience utility for quickly parsing a small snippet of text using a particular language and getting access to its root node. It is meant for demonstration purposes. If you are going to need to reparse the text after an edit has been made, you should create a full parser with `parser()` and use `parser_parse()` instead.

Usage

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

Arguments

x	[string] The text to parse.
language	[tree_sitter_language] The language to parse with.

Value

A root node.

Examples

```
language <- treesitter.r::language()
text <- "map(xs, function(x) 1 + 1)"

# Note that this directly returns the root node, not the tree
text_parse(text, language)
```

---

tree-accessors *Tree accessors*

Description

- `tree_text()` retrieves the tree's `text` that it was parsed with.
- `tree_language()` retrieves the tree's `language` that it was parsed with.
- `tree_included_ranges()` retrieves the tree's `included_ranges` that were provided to `parser_set_included_range`. Note that if no ranges were provided originally, then this still returns a default that always covers the entire document.

Usage

```
tree_included_ranges(x)

tree_text(x)

tree_language(x)
```

Arguments

x	[tree_sitter_tree] A tree.

Value

- `tree_text()` returns a string.
- `tree_language()` returns a `tree_sitter_language`.
- `tree_included_ranges()` returns a list of `range()` objects.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)

tree_text(tree)
tree_language(tree)
tree_included_ranges(tree)
```

`TreeCursor` *Tree cursors*

Description

`TreeCursor` is an R6 class that allows you to walk a tree in a more efficient way than calling `node_*()` functions like `node_child()` repeatedly.

You can also more elegantly create a cursor with `node_walk()` and `tree_walk()`.

Value

R6 object representing the tree cursor.

Methods

Public methods:

- `TreeCursor$new()`
- `TreeCursor$reset()`
- `TreeCursor$node()`
- `TreeCursor$field_name()`
- `TreeCursor$field_id()`
- `TreeCursor$descendant_index()`
- `TreeCursor$goto_parent()`
- `TreeCursor$goto_next_sibling()`
- `TreeCursor$goto_previous_sibling()`
- `TreeCursor$goto_first_child()`
- `TreeCursor$goto_last_child()`
- `TreeCursor$depth()`
- `TreeCursor$goto_first_child_for_byte()`
- `TreeCursor$goto_first_child_for_point()`

Method `new()`**:** Create a new tree cursor.

Usage:

```
TreeCursor$new(node)
```

Arguments:

`node [tree_sitter_node]`
The node to start walking from.

Method `reset()`**:** Reset the tree cursor to a new root node.

Usage:

```
TreeCursor$reset(node)
```

Arguments:

`node [tree_sitter_node]`
The node to start walking from.

Method `node()`**:** Get the current node that the cursor points to.

Usage:

```
TreeCursor$node()
```

Method `field_name()`**:** Get the field name of the current node.

Usage:

```
TreeCursor$field_name()
```

Method `field_id()`**:** Get the field id of the current node.

Usage:

```
TreeCursor$field_id()
```

Method `descendant_index()`**:** Get the descendent index of the current node.

Usage:

```
TreeCursor$descendant_index()
```

Method `goto_parent()`**:** Go to the current node's parent.
Returns `TRUE` if a parent was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_parent()
```

Method `goto_next_sibling()`**:** Go to the current node's next sibling.
Returns `TRUE` if a sibling was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_next_sibling()
```

Method `goto_previous_sibling()`**:** Go to the current node's previous sibling.
Returns `TRUE` if a sibling was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_previous_sibling()
```

Method `goto_first_child()`**:** Go to the current node's first child.
Returns `TRUE` if a child was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_first_child()
```

Method `goto_last_child()`**:** Go to the current node's last child.

Returns `TRUE` if a child was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_last_child()
```

Method `depth()`**:** Get the depth of the current node.

Usage:

```
TreeCursor$depth()
```

Method `goto_first_child_for_byte()`**:** Move the cursor to the first child of its current node that extends beyond the given byte offset.

Returns `TRUE` if a child was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_first_child_for_byte(byte)
```

Arguments:

`byte [double(1)]`
: The byte to move the cursor past.

Method `goto_first_child_for_point()`**:** Move the cursor to the first child of its current node that extends beyond the given point.

Returns `TRUE` if a child was found, and `FALSE` if not.

Usage:

```
TreeCursor$goto_first_child_for_point(point)
```

Arguments:

`point [tree_sitter_point]`
: The point to move the cursor past.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function(a, b) { a + b }"

tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

cursor <- TreeCursor$new(node)

cursor$node()
cursor$goto_first_child()
cursor$goto_first_child()
cursor$node()
cursor$goto_next_sibling()
cursor$node()
```

`tree_root_node` *Retrieve the root node of the tree*

Description

`tree_root_node()` is the entry point for accessing nodes within a specific tree. It returns the "root" of the tree, from which you can use other `node_*()` functions to navigate around.

Usage

```
tree_root_node(x)
```

Arguments

`x`	`[tree_sitter_tree]`
	A tree.

Value

A node.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)
node <- tree_root_node(tree)

# Trees and nodes have a similar print method, but you can
# only use other `node_*()` functions on nodes.
tree
node

node |>
  node_child(1) |>
  node_children()
```

`tree_root_node_with_offset` *Retrieve an offset root node*

Description

`tree_root_node_with_offset()` is similar to `tree_root_node()`, but the returned root node's position has been shifted by the given number of bytes, rows, and columns.

This function allows you to parse a subset of a document with `parser_parse()` as if it were a self-contained document, but then later access the syntax tree in the coordinate space of the larger document.

Note that the underlying `text` within `x` is not what you are offsetting into. Instead, you should assume that the `text` you provided to `parser_parse()` already contained the entire subset of the document you care about, and the offset you are providing is how far into the document the beginning of `text` is.

Usage

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

Arguments

`x`	`[tree_sitter_tree]` A tree.
`byte, point`	`[double(1), tree_sitter_point]` A byte and point offset combination.

Value

An offset root node.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "fn <- function() { 1 + 1 }"
tree <- parser_parse(parser, text)

# If `text` was the whole document, you can just use `tree_root_node()`
node <- tree_root_node(tree)

# If `text` represents a subset of the document, use
# `tree_root_node_with_offset()` to be able to get positions in the
# coordinate space of the original document.
byte <- 5
point <- point(5, 0)
```

```
node_offset <- tree_root_node_with_offset(tree, byte, point)

# The position of `fn` if you treat `text` as the whole document
node |>
  node_child(1) |>
  node_child(1)

# The position of `fn` if you treat `text` as a subset of a larger document
node_offset |>
  node_child(1) |>
  node_child(1)
```

`tree_walk` *Generate a* `TreeCursor` *iterator*

Description

`tree_walk()` creates a TreeCursor starting at the root node. You can use it to "walk" the tree more efficiently than using `node_child()` and other similar node functions.

Usage

```
tree_walk(x)
```

Arguments

`x`	`[tree_sitter_tree]` A tree.

Value

A `TreeCursor` object.

Examples

```
language <- treesitter.r::language()
parser <- parser(language)

text <- "1 + foo"
tree <- parser_parse(parser, text)

cursor <- tree_walk(tree)

cursor$goto_first_child()
cursor$goto_first_child()
cursor$node()
cursor$goto_next_sibling()
cursor$node()
```

Index