Package ‘parsetools’

April 8, 2020

Type Package

Title Parse Tools

Version 0.1.3

Maintainer Andrew Redd <andrew.redd@hsc.utah.edu>

License GPL-2

Language en-US

Imports methods, utils

Suggests covr, knitr, rmarkdown, testthat

Description Tools and utilities for dealing with parse data.
Parse data represents the parse tree as data with location and type
information. This package provides functions for navigating the
parse tree as a data frame.

Collate 'internal.R' 'accessors.R' 'checks.R' 'children.R'
'comments.R' 'errors.R' 'family.R' 'find-utils.R' 'firstborn.R'
'get_parse_data.R' 'grouping.R' 'iff_blocks.R' 'parent.R'
'pd_assign.R' 'pd_call.R' 'pd_make_is_in.R' 'pd_classes.R'
'pd_comments.R' 'pd_function.R' 'pd_identify.R' 'pd_if.R'
'reconstitute.R' 'root.R' 'siblings.R' 'tags.R'
'testing_blocks.R' 'zzz.R'

Encoding UTF-8

RoxygenNote 7.0.2

VignetteBuilder knitr

KeepSource yes

URL https://github.com/RDocTaskForce/parsetools

BugReports https://github.com/RDocTaskForce/parsetools/issues

NeedsCompilation no

Author Andrew Redd [aut, cre],
R Consortium [cph, fnd]

Repository CRAN

Date/Publication 2020-04-08 16:30:02 UTC

R topics documented:

`all_grouping_ids` *get the grouping ids*

Description

get the ids that represent the grouping nodes.

Usage

```
all_grouping_ids(pd = get("pd", parent.frame()))
```

Arguments

`pd`	The `parse-data` information

Value

an integer vector of ids.

`all_root_nodes` *Find all root node from parse data*

Description

A root node in a file is a standalone expression, such as in source file a function definition. when discussing a subset it is any expression that does not have a parent in the subset.

Usage

```
all_root_nodes(pd, include.groups = TRUE)
```

Arguments

`pd`	The `parse-data` information
`include.groups`	descend into grouped code {}?

Value

`parse-data` with for the root nodes.

`assignments` *Assignment Node Navigation.*

Description

These function help identify and navigate assignments in parse data.

Usage

```
pd_is_assignment(id, pd, .check = TRUE)

pd_get_assign_value_id(id, pd, .check = TRUE)

pd_get_assign_variable_id(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Details

These functions only deal with assignment operators. Using `base::assign()` or `base::delayedAssign()` are considered calls in terms of parse data.

There are five assignment operators grouped into three categories.

- Left assignment, the `<-` and `<<-`,
- right assignment, `->` and the rarely used `->>`
- and the equals assignment `=`.

Functions

- `pd_is_assignment`: Check if the node is an assignment expression.
- `pd_get_assign_value_id`: Get the id for the value portion of an assignment.
- `pd_get_assign_variable_id`: Get the variable of an assignment.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

# The first should be an assignment
pd_is_assignment(roots[[1]], pd=pd)

# the variable/value of the assignment can be accessed by
variable.id <- pd_get_assign_variable_id(roots[[1]], pd)
value.id <- pd_get_assign_value_id(roots[[1]], pd)
# Note that these function will give the variable/value part
# for both LEFT_ASSIGN and RIGHT_ASSIGN operators, going by order
# of ids, or position in the data may not give the expected results.
```

`calls` *Call nodes*

Description

Call nodes represent function calls.

Retrieves the ids of the arguments of a call as an integer vector.

Usage

```
pd_is_call(id, pd, calls = NULL, .check = TRUE)

pd_is_symbol_call(id, pd, .check = TRUE)

pd_get_call_symbol_id(id, pd, .check = TRUE)

pd_get_call_arg_ids(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`calls`	an optional list of calls to restrict consideration to.
`.check`	Perform checks for input validation?

Details

The traditional call of `function_name(arguments)` is a symbol call as `function_name` is the symbol directly referencing the function to call. Other calls may exists such as `function_array[[1]]()` which first indexes the `function_array` then calls the returned function. This qualifies as a call expression but not a symbol call expression. We are often only concerned with symbol calls and not the anonymous version.

Value

a logical of the same length as `id`

a named list where each element is the id for the `expr` element of the argument.

Functions

- `pd_is_call`: Test if the node is a call expression.
- `pd_is_symbol_call`: Test if the node is specifically a symbol call expression.
- `pd_get_call_symbol_id`: Get the symbol, i.e. the name of the function, being called.
- `pd_get_call_arg_ids`: test Get the set of arguments to the function call.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

# which root is a call?
pd_is_call(roots, pd)
id <- roots[pd_is_call(roots, pd)]
# not all calls are symbole calls.
pd_is_symbol_call(id, pd)
# what is the symbol being called?
pd_text(pd_get_call_symbol_id(id, pd), pd)
# what are the arguments to the call
args <- pd_get_call_arg_ids(id, pd)
pd_token(pd_get_firstborn(args, pd), pd)
pd_text(pd_get_firstborn(args, pd), pd)
```

`clean_tag_comments` *clean tag comments*

Description

replaces '`#@tag`' with '`#! @tag`'

Usage

```
clean_tag_comments(x, tag)
```

Arguments

`x`	text to strip from
`tag`	tag(s) to remove

`extract_test_block` *Extract testing blocks from the parse-data.*

Description

Extract the content of a testing block as a character vector of lines. The name, which is attached as an attribute is taken from the info string or inferred by location, see Details.

Usage

```
extract_test_block(
  id = all_tagged_iff_block_ids(pd, .testing.tags),
  pd = get("pd", parent.frame())
)
```

Arguments

`id`	iff block id, not the content
`pd`	a parse-data object.

Value

a character vector with the lines for the specific test(s) with the name of the test included as an attribute.

`extract_test_blocks` *extract tests from a file.*

Description

Convenience function for extracting all tests from a file. This parses the file and passes the work to `extract_test_block`.

Usage

```
extract_test_blocks(file)
```

Arguments

`file`	the file to retrieve tests from.

`family-nodes` *Family-wise Node Identification and Navigation.*

Description

Parse data is organized into a hierarchy of nodes. These functions provide simple ways to identify the nodes of interest, often from a specified node of interest.

Test if an expression is the firstborn, i.e. oldest or lowest id.

Usage

```
pd_get_children_ids(
  id,
  pd,
  ngenerations = 1,
  include.self = FALSE,
  aggregate = TRUE,
  .check = TRUE
)

pd_is_firstborn(id, pd, .check = TRUE)

pd_get_firstborn(id, pd, .check = TRUE)

pd_get_parent_id(id, pd, .check = TRUE)

pd_get_ancestor_ids(
  id,
  pd,
  ngenerations = Inf,
  aggregate = TRUE,
  include.self = TRUE,
  only.present = FALSE,
  last = 0L,
  .check = TRUE
)

pd_get_sibling_ids(id, pd, .check = TRUE)

pd_get_next_sibling_id(id, pd, .check = TRUE)

pd_get_prev_sibling_id(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`ngenerations`	Number of generations to go forwards or backwards.
`include.self`	Should the root node (`id`) be included?
`aggregate`	Should aggregate(TRUE) or only the the final (FALSE) generation be returned?
`.check`	Perform checks for input validation?
`only.present`	should the list be restricted to only those node that are present? Most relevant for when parent is zero.
`last`	The last acceptable parent.

Details

The language parsetools uses is that of family. Similar to a family each node could have: a *parent*, the node that contains the node in question; *children*, the nodes contained by the given node; *ancestors*, the collection of nodes that contain the given node, it's parent, it's parent's parent, and so on; and *descendents*, the collection of nodes that are contained by the given node or contained by those nodes, and so on. Terminology is analogous, a *generation* is all the the nodes at the same depth in the hierarchy. A node may have *siblings*, the set of nodes with the same parent. If a node does not have a parent it is called a *root* node.

Similarly, age is also used as an analogy for ease of navigation. Generally, nodes are numbered by the order that they are encountered, when parsing the source. Therefore the node with the smallest `id` among a set of siblings is referred to the *firstborn*. This is give the special designation as it is the most often of children used, as it likely determines the type of call or expression that is represented by the node. The firstborn has no 'older' siblings, the 'next' sibling would be the next oldest, i.e. the node among siblings with the smallest id, but is not smaller that the reference node id.

In all cases when describing function the `id`, is assumed to be in the context of the parse data object `pd` and for convenience refers to the node associated with said `id`.

Functions

- `pd_get_children_ids`: Get all nodes that are children of `id`. Get all ids in pd that are children of `id`. i.e. lower in the hierarchy or with id as a parent. If `ngenerations` is greater than 1 and `aggregate` is `TRUE`, all descendents are aggregated and returned.
- `pd_is_firstborn`: Test if `id` is firstborn.
- `pd_get_firstborn`: Get the firstborn child of `id`.
- `pd_get_parent_id`: Get the parent of `id`.
- `pd_get_ancestor_ids`: Get the ancestors of `id`.
- `pd_get_sibling_ids`: Identify siblings of `id`.
- `pd_get_next_sibling_id`: Get the next younger sibling.
- `pd_get_prev_sibling_id`: Get the next older sibling.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

# assignments have three children
# The operator, the assignment, and the value.
kids <- pd_get_children_ids(roots[[1]], pd)
# The token tells what kind of node the ids represent.
pd_token(kids, pd)
```

`function-nodes` *Function Nodes*

Description

These function help identify and navigate noses associated with function definition.

Usage

```
pd_is_function(id, pd, .check = TRUE)

pd_is_in_function(id, pd, .check = TRUE)

pd_get_function_body_id(id, pd, .check = TRUE)

pd_get_function_arg_ids(id, pd, .check = TRUE)

pd_get_function_arg_variable_ids(id, pd, .check = TRUE)

pd_get_function_arg_variable_text(id, pd, .check = TRUE)

pd_is_function_arg(id, pd, .check = TRUE)

pd_get_function_arg_associated_comment_ids(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Details

A function node is the node for the expression that has as it's children the function keyword(firstborn), the arguments, including the nodes representing the opening closing parentheses in the definition, and finally a node, as the youngest, for the body of the function.

Functions

- `pd_is_function`: Test if the `id` points to a function.
- `pd_is_in_function`: test if a node is contained in a function definition.
- `pd_get_function_body_id`: Obtain the body of a function
- `pd_get_function_arg_ids`: Obtain the ids for the arguments of a function
- `pd_get_function_arg_variable_ids`: Retrieve the variable for a function argument
- `pd_get_function_arg_variable_text`: Get the variable names for a function definition.

- `pd_is_function_arg`: is `id` a function argument?
- `pd_get_function_arg_associated_comment_ids`: Retrieve relative documentation comments associated with function arguments.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

function.id <- pd_get_assign_value_id(roots[[1]], pd)
pd_is_function(function.id, pd)
length(function.kids <- pd_get_children_ids(function.id, pd))
# function nodes have many because it contains
# 1. the function keyword.
# 2. the parentheses '(' and ')'
# 3. each argument name plus the equals sign and value, if given.
# 4. and finally, and expr node for the function body.
pd_token(function.kids, pd)
# even though there are only two argument since each has
# a default value given there are 6 total nodes that
# return true as function arguments, care is needed when
# dealing with function arguments.
pd_is_function_arg(function.kids, pd)
pd_get_function_arg_ids(function.id, pd)
# A simple way to identify the argument names is
pd_get_function_arg_variable_text(function.id, pd)

# To identify the function body node.
pd_get_function_body_id(function.id, pd)
```

`get_family_pd` *Get family of nodes.*

Description

Subset the pd to the family of `id`.

Usage

```
get_family_pd(
  id,
  pd,
  include.self = TRUE,
  ngenerations = Inf,
  ...,
  include.doc.comments = TRUE,
  include.regular.comments = FALSE
)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`include.self`	Should the root node (`id`) be included?
`ngenerations`	Number of generations to go forwards or backwards.
`...`	currently ignored.
`include.doc.comments`	include associated documentation comments.
`include.regular.comments`	include associated regular comments.

Value

a subset of the `parse-data` pd.

`get_parse_data` *Parse Data*

Description

Parsing data is at the core of parse tools and thus at the core of the documentation package. The `get_parse_data` function is essentially a customized version of <getParseData> that will return a cleaned up version of the parse data for a variety of objects. This version also fails less often, even reparsing text when needed.

valid_parse_data: The `valid_parse_data` function tests if the object `df` conforms to the expected conventions of a `parse-data` object. Returns TRUE if valid otherwise returns the reason it is not valid.

as_parse_data: The `as_parse_data` function tests if a data frame is valid through `valid_parse_data` then returns the data with the comments classified, as is expected for parse-data objects. All parse data for use with parsetools functions should be obtained either through get_parse_data or converted through as_parse_data.

Usage

```
get_parse_data(x, ...)

## S3 method for class 'srcfile'
get_parse_data(x, ...)

## S3 method for class 'srcref'
get_parse_data(
  x,
  ...,
  ignore.groups = TRUE,
  include.doc.comments = TRUE,
  include.regular.comments = FALSE
)

## S3 method for class '`function`'
get_parse_data(x, ...)

valid_parse_data(df)

as_parse_data(df)
```

Arguments

`x`	an object to get parse-data from.
`...`	options for specific type of objects.
`ignore.groups`	Should groupings be ignored?
`include.doc.comments`	include associated documentation comments.
`include.regular.comments`	include associated regular comments.
`df`	a data.frame object.

Methods (by class)

- `function`: Get parse information from a function. The function must have a `srcref`.

Examples

```
text <- "    my_function <- function(object #< An object to do something with
        ){
    #' A title
    #'
    #' A Description
    print(\"It Works!\")
    #< A return value.
}"
source(textConnection(text), keep.source = TRUE)
```

```
# Get parse data from a function
(pd <- get_parse_data(my_function))
# which must have a srcref attribute.
# You can call the get_parse data directly on the srcref object.
src <- utils::getSrcref(my_function)
pd2 <- get_parse_data(src)

identical(pd, pd2)

# Objects must have a srcref.
utils::getSrcref(rnorm)
tools::assertError(get_parse_data(rnorm), verbose = TRUE)
```

`if-statements` *If Statement Nodes*

Description

These function navigate logic statements.

Returns the id of the predicate of the if statement, i.e. the conditional statement.

Returns the id of the body of the branch executed if the predicate evaluates to true.

Gets the id of the alternate branch, i.e. the else branch.

Usage

```
pd_is_if(id, pd, .check = TRUE)

pd_get_if_predicate_id(id, pd, .check = TRUE)

pd_get_if_branch_id(id, pd, .check = TRUE)

pd_get_if_alternate_id(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Details

If statements have the form of the following.

```
if (predicate) branch else alternate
```

The `predicate` refers to the logical test being performed. The `branch` is the statement or block that is executed if `predicate` evaluates true. The `alternate` is the statement of block that is executed if `predicate` returns false.

Value

an id integer.

an id integer.

Functions

- `pd_is_if`: Is node an if expression.
- `pd_get_if_predicate_id`: Get the predicate node.
- `pd_get_if_branch_id`: Get the `branch` statement or block node.
- `pd_get_if_alternate_id`: Get the `alternate` statement or block node.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

# Find the if statement
is.if <- pd_is_if(pd$id, pd=pd)
sum(is.if)
if.id <- pd$id[is.if]

# The predicate
pd_reconstitute(pd_get_if_predicate_id(if.id, pd), pd)

# The branch for if predicate evaluates TRUE
pd_reconstitute(pd_get_if_branch_id(if.id, pd), pd)

# The alternate for if predicate evaluates FALSE
pd_reconstitute(pd_get_if_alternate_id(if.id, pd), pd)
```

`iff-blocks` *IFF Blocks*

Description

IFF is short for if(FALSE)\{#@tag ... blocks. These block can contain development, testing, or example code that can be extracted into documentation or other files.

Usage

```
pd_is_iff(id, pd, allow.short = TRUE, .check = TRUE)

pd_is_iff_block(id, pd, allow.short = TRUE, .check = TRUE)

pd_all_iff_ids(pd = get("pd", parent.frame()), ...)

pd_all_iff_block_ids(pd, root.only = TRUE, ignore.groups = FALSE, ...)

pd_is_tagged_iff_block(id, pd, tag, doc.only = TRUE, ..., .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`allow.short`	if `if(F)` should be considered an IFF block.
`.check`	Perform checks for input validation?
`...`	passed along.
`root.only`	only root blocks(`TRUE`) or all block (`FALSE`)
`ignore.groups`	Ignore code grouping
`tag`	The tag to consider.
`doc.only`	Should comments be restricted to documentation style comments only?

Details

Here are some examples:

- if(FALSE)\{#' @test ... Is valid and tags the block as a test.
- if(FALSE)\{#@test ... Is valid and tags the block as a test. Note here that we are using the #@ tag comment.
- if(FALSE)\{# @test ... Is valid only if `doc.only==FALSE`.
- if(FALSE)#@test ...

Functions

- `pd_is_iff`: This function tests if an expression id is the root of an `if(FALSE)` statement, differs from `pd_is_iff_block` in that it will return TRUE even if the conditional statement is not a formal bracketed block `{...}`.
- `pd_is_iff_block`: Tests if an expression id is the root of an `if(FALSE)` block statement, differs from `pd_is_iff` in that in addition to it being an if(FALSE) expression the conditional branch of the logic must be a braced block of code. E.g. if given the id corresponding to `if(FALSE){...}`, both `pd_is_iff()` and `pd_is_iff_block()` would return TRUE while for `if(FALSE)do_somthing()` `pd_is_iff()` would return TRUE but `pd_is_iff_block()` would return FALSE because the expression is not a 'block' statement.
- `pd_all_iff_ids`: Get all ids corresponding to IFF expressions.

- `pd_all_iff_block_ids`: Get all ids corresponding to IFF block
- `pd_is_tagged_iff_block`: Test if a block if both an IFF block statement and is tagged. To tag an IFF block the first pared element must be a comment that contains an '@' tag to denote a special block. The comment on the same line as the opening brace or on any subsequent line but cannot be preceded by any other statement.

`internal` *Make a function operate internal to parsetools*

Description

Convert a function to look for pd object in the `parent.frame()`, and the id to extract from the pd unless overwritten.

These functions are for internal use but are documented here for reference.

Usage

```
internal(fun, id = pd$id)

token(id = pd$id, pd = get("pd", parent.frame()))

text(id = pd$id, pd = get("pd", parent.frame()))

nodes(id, pd = get("pd", parent.frame()))

start_line(id, pd = get("pd", parent.frame()))

start_col(id, pd = get("pd", parent.frame()))

end_line(id, pd = get("pd", parent.frame()))

end_col(id, pd = get("pd", parent.frame()))

filename(pd = get("pd", parent.frame()))

lines(id, pd = get("pd", parent.frame()))

is_terminal(id, pd = get("pd", parent.frame()))

is_first_on_line(id, pd = get("pd", parent.frame()))

is_last_on_line(id, pd = get("pd", parent.frame()))

spans_multiple_lines(id, pd = get("pd", parent.frame()))

terminal_ids_on_line(line, pd = get("pd", parent.frame()))
```

```
ids_starting_on_line(line, pd = get("pd", parent.frame()))

ids_ending_on_line(line, pd = get("pd", parent.frame()))

prev_terminal(id = pd$id, pd = get("pd", parent.frame()))

expr_text(id, pd = get("pd", parent.frame()))
```

Arguments

`fun`	The function to make internal
`id`	the ID of the expression
`pd`	the parse data.
`line`	a line number

Functions

- `token`: Extract the token
- `text`: Extract the text
- `nodes`: Extract only the specified node(s).
- `start_line`: Get the line the expression starts on.
- `start_col`: Get the column the expression starts on.
- `end_line`: Get the line the expression ends on.
- `end_col`: Get the column the expression ends on.
- `filename`: Extract the filename if available, otherwise return "<UNKNOWN>".
- `lines`: Extract the lines of text.
- `is_terminal`: does id represent a terminal node.
- `is_first_on_line`: is an expression the first one on a line?
- `is_last_on_line`: Is expression the last terminal node on the line?
- `spans_multiple_lines`: does the expression span multiple lines?
- `terminal_ids_on_line`: Get the ids on a given line that are terminal nodes.
- `ids_starting_on_line`: Get ids for nodes that start on the given line
- `ids_ending_on_line`: Get ids for nodes that end on the given line
- `prev_terminal`: Get the id for the terminal expression that is immediately prior to the one given.
- `expr_text`: If id represents an `expr` token reiterate on the firstborn. Throws an error if anything but an expression or text if found.

`n_children` *Count the number of children*

Description

Count the number of children

Usage

```
n_children(id = pd$id, pd = get("pd", parent.frame()))
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information

`pd_all_tagged_iff_block_ids` *Find all tagged* `if(FALSE)` *blocks.*

Description

Retrieves all ids identifying `if(FALSE)` blocks that are also tagged with `tag`. See `pd_is_tagged_iff_block` for details.

Usage

```
pd_all_tagged_iff_block_ids(pd, tag, doc.only = TRUE)
```

Arguments

`pd`	The `parse-data` information
`tag`	The tag to consider.
`doc.only`	Should comments be restricted to documentation style comments only?

Value

an integer vector giving the ids in pd that identify `if(FALSE)`blocks that are also tagged with `tag`.

See Also

`pd_is_iff_block`, `pd_is_tagged_iff_block`, `pd_has_tag`

pd_class_definitions *Test for Class Definitions*

Description

These function manage adding class defining functions and testing if an id is associated with a class definition or if is contained in the class definition.

Usage

```
pd_class_definitions

pd_is_class_definition(id, pd, .check = TRUE)

pd_is_in_class_definition(id, pd, .check = TRUE)

pd_add_class_definition(
  name,
  test.is,
  test.in,
  .exists = TRUE,
  .overwrite = FALSE
)

pd_add_class(name, .exists = TRUE, .overwrite = FALSE)
```

Arguments

`id`	id(s) to test.
`pd`	parse data which contains id.
`.check`	should the id, and pd be checked?
`name`	name of the class defining function
`test.is`	function accepting arguments `id` and `pd` which tests if given `id` is associated with the defined class defining functions.
`test.in`	function accepting arguments `id` and `pd` which tests if given `id` is contained in the defined class defining functions.
`.exists`	require the function to exists to add.
`.overwrite`	if TRUE allows for overwriting existing test functions.

Format

An object of class `environment` of length 12.

Details

`pd_class_definitions$has`**:**

Usage:

```
        pd_class_definitions$has(name)
```

Check if a class defining function has 'is' and 'in' function defined for it.

`pd_class_definitions$add` **or** `pd_add_class`**:**

Usage:

```
pd_class_definitions$add(name, .exists=TRUE, .overwrite=FALSE)

pd_add_class(name, .exists=TRUE, .overwrite=FALSE)
```

Add a def with default 'is' and 'in' functions defined.

`pd_class_definitions$add_definition` **or** `pd_add_class_definition`**:**

Usage:

```
pd_class_definitions$add_definition(name, test.is, test.in, .exists=TRUE, .overwrite=FALSE)

pd_add_class_definition(name, test.is, test.in, .exists=TRUE, .overwrite=FALSE)
```

Add a class defining function with custom 'is' and 'in' functions defined.

`pd_class_definitions$rm`**:**

Usage:

```
pd_class_definitions$rm(name)
```

Remove the testing functions for the class.

`pd_class_definitions$names`**:**

Usage:

```
pd_class_definitions$names()
```

Return a vector of the classed for which tests are defined.

`pd_class_definitions$test_is`**:**

Usage:

```
pd_class_definitions$test_is(id, pd, check=TRUE)
```

Test if `id` is associated with each of defined class definitions.

`pd_class_definitions$test_is_in`**:**

Usage:

```
pd_class_definitions$test_is_in(id, pd, check=TRUE)
```

Test if `id` is contained within each of defined class definitions.

`pd_class_definitions$which`**:**

Usage:

```
pd_class_definitions$which(id, pd, check=TRUE)
```

Return the name of the class, if any, which `id` corresponds to.

`pd_class_definitions$in_which`:

Usage:

```
pd_class_definitions$in_which(id, pd, check=TRUE)
```

Returns a vector of the classes, if any, of the classes which `id` is contained in.

`pd_is_class_definition`: Returns `TRUE` if the id corresponds to any of the class defining calls.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

# Get the 'setClass' call.
class.id <- pd_get_assign_value_id(roots[2], pd)
# Check to make sure that it is a function that sets a class.
pd_is_class_definition(class.id, pd)
# and that it is the setClass call.
pd_text(pd_get_call_symbol_id(class.id, pd), pd)
```

`pd_get_closest_call_id` *Get the closest call ID.*

Description

Get the id of the call that is closest to the `id` given. Closest is defined as the innermost call that contains the `id`.

Usage

```
pd_get_closest_call_id(id, pd, calls = NULL, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`calls`	optional calls to limit consideration to.
`.check`	Perform checks for input validation?

`pd_get_comment_tag_content` *Get the content of a tag*

Description

Get the content of a tag

Usage

```
pd_get_comment_tag_content(id, pd, tag, all.contiguous = FALSE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`tag`	tag(s) to test for
`all.contiguous`	if TRUE get all comments connected to this element.

`pd_get_iff_associated_name_id` *Find the name that should be associated with an* `if(FALSE)` *block.*

Description

For `if(FALSE)` documentation blocks, such as `@testing` and `@example` blocks, a user may supply an information string which gives the name information for tests and examples. for example, in if(FALSE)\{#@test my special test the information string is "my special test".

The more common case is when there is no information string. In these cases the name is inferred by the previous assignment or declaration.

The `id` argument should identify one and only one `if(FALSE)` block, but as this is an internal function, argument checks are not performed.

Usage

```
pd_get_iff_associated_name_id(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Details

IFF blocks can be placed sequentially and `pd_get_iff_associated_name_id` will navigate back until it finds a non-IFF block to use for the name. This way users can place multiple tests and examples after a declaration.

If the previous expression is an assignment, the assignee variable of the assignment is chosen as the name. An attribute 'type' is also set on the return value. For function assignments `type="function_assignment"`, for all other assignments `type="assignment"`.

The names for `link{setClass}` calls will also be inferred. The name of the class is taken as the name, but the return value also has the attribute of `type="setClass"`. Note that it is common to assign the result of `setClass` to a variable, which may or may not match the class name. In those cases the assignment operation takes priority and would have `type="assignment"`.

The names for `setMethod` will assume the S3 convention of `<method>.<class>`. In the case the the signature is more than just the class, the signature will be collapsed, separated by commas. the type attribute will be set to `"setMethod"`.

`setGeneric` can also be used with the name of the generic function the inferred name and `type="setGeneric"`. `setAs` infers coerce methods. `type="setAs"`.

`pd_get_relative_comment_associated_ids` *Associate relative documentation comments*

Description

Relative comment created with `#\<` comment tags document something designated by the location of the comment. In general, the comment documents the previous symbol. A comment will not be associated with any parse id that does not have the same parent as the comment. For example,

Usage

```
pd_get_relative_comment_associated_ids(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Details

```
function(x #< a valid comment
        ){}
```

would associate a `valid comment` with `x`, but

```
function(x){ #< not a valid comment
           }
```

would not.

Value

Returns a vector of the same length as id. Where the value is either the id of the associated object or NA if it cannot be associated.

`pd_get_tagged_comment_ids` *Get tagged comment ids*

Description

Finds all ids that are comments and contain the given '@' `tag`. If doc.only is true(default) then only documentation comments are considered, otherwise all comments are examined.

Usage

```
pd_get_tagged_comment_ids(pd, tag, doc.only = TRUE)
```

Arguments

`pd`	The `parse-data` information
`tag`	tag(s) to test for
`doc.only`	Restrict to documentation comments only?

Value

an integer vector of ids.

`pd_has_tag` *Check if there is a documentation `@` tag.*

Description

Check if a node of `parse-data` identified by `id` is both a comment and contains a documentation tag identified by the `@` symbol.

Usage

```
pd_has_tag(id, pd, tag, ...)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`tag`	tag(s) to test for
`...`	options passed on

pd_identify *Get the ID for an object*

Description

Identify in pd the id for the object given.

Usage

```
pd_identify(pd, object)

## Default S3 method:
pd_identify(pd, object)

## S3 method for class '`NULL`'
pd_identify(pd, object)

## S3 method for class 'srcref'
pd_identify(pd, object)
```

Arguments

pd	the parse data.
object	an object that originated in pd, for which to obtain the ID.

Methods (by class)

- `default`: Default method identifies by `base::srcref()`.
- `NULL`: Passing a NULL object will result in an error.
- `srcref`: Identify by explicit `srcref`.

pd_is_comment *Is this a comment?*

Description

pd_is_comment: Test if an id represents a comment of any kind.

pd_is_relative_comment: Tests if the comment is a relative (location dependent) type comment.

pd_all_relative_comment_ids: Retrieve all ids associated with relative comments.

pd_is_doc_comment: Additionally tests if the comment is a documentation type comment.

Usage

```
pd_is_comment(id, pd, .check = TRUE)

pd_is_relative_comment(id, pd, .check = TRUE)

pd_all_relative_comment_ids(pd)

pd_is_doc_comment(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

Value

Should return a logical vector, for parse-data and data.frame should be length of `nrow(x)`. For character same length as x.

`pd_is_grouping` *test if an id is a grouping element*

Description

A grouping is defined as a non empty set starting with a curly brace token and and for which there is no parent or the parent is also a grouping.

Usage

```
pd_is_grouping(id, pd, .check = TRUE)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`.check`	Perform checks for input validation?

pd_make_is_in_call *Create a function to test if an id is contained in a type*

Description

Create a function to test if an id is contained in a type

Usage

```
pd_make_is_in_call(calls, .is = pd_make_is_call(calls))

pd_make_is_call(calls)
```

Arguments

calls	The tokens to test against
.is	A function to test if a specific id is a valid

pd_reconstitute *Reconstitute Expressions*

Description

Creates expressions and calls from the given `id` and parse-data, pd.

Usage

```
pd_reconstitute(id, pd, .check = TRUE)
```

Arguments

id	id of the expression of interest
pd	The `parse-data` information
.check	Perform checks for input validation?

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)

pd_reconstitute(roots[1], pd)
```

`pd_text` *Accessor functions*

Description

This collection of function can be used to easily access elements of the parse data information.

Usage

```
pd_text(id, pd)
```

Arguments

`id`	the ID of the expression
`pd`	the parse data.

`root` *Root IDs*

Description

Root IDs constitute the id of a stand alone expression. That is one that is not contained inside of another call or expression. The one exception to this is code blocks denoted by curly braces that are not themselves part of another call or expression; these we call code groups. In definition, A root node is defined to be a node that either has no parent or whose parent is a grouping node.

Usage

```
pd_is_root(id, pd, ignore.groups = TRUE, .check = TRUE)

pd_all_root_ids(pd, include.groups = TRUE)

ascend_to_root(
  id = pd$id,
  pd = get("pd", parent.frame()),
  ignore.groups = TRUE,
  .check = TRUE
)
```

Arguments

`id`	id of the expression of interest
`pd`	The `parse-data` information
`ignore.groups`	Should groupings be ignored?
`.check`	Perform checks for input validation?
`include.groups`	Include groups as root nodes (T) or descend into groups for roots?

Details

If `ignore.groups=TRUE` then groupings are ignored and root nodes within the group are interpreted as roots, otherwise nodes within a group are not interpreted as root. Groupings are always interpreted as root if the parent is 0 or if the parent is a group and also a root.

Functions

- `pd_is_root`: Test if a node is a root node
- `pd_all_root_ids`: give all root ids in pd
- `ascend_to_root`: ascend from id to root

See Also

see `pd_is_grouping` for details on what a grouping is.

Examples

```
# load example file and get_parse data
ex.file <- system.file("examples", "example.R", package="parsetools")
exprs <- parse(ex.file, keep.source = TRUE)
pd <- get_parse_data(exprs)

# There are 3 expressions so there should be three roots.
sum(pd_is_root(pd$id, pd))
roots <- pd_all_root_ids(pd)
```

`strip_doc_comment_leads` *Remove the characters identifying a documentation comment.*

Description

Remove the characters identifying a documentation comment as a document comment leaving only the relevant text.

Usage

```
strip_doc_comment_leads(comment, rm.space = TRUE)
```

Arguments

`comment`	The text of the comments or parse data.
`rm.space`	should the space at the beginning of the line be removed.

`strip_tag` *Remove a tag that identified a line.*

Description

Removes `@tag` tags from the text. Also will remove '`#@tag`' replacing with '`#!`'.

Usage

```
strip_tag(x, tag, ...)
```

Arguments

`x`	text to strip from
`tag`	tag(s) to remove
`...`	passed on options]

Index