Package 'lobstr'

May 8, 2026

Title Visualize R Data Structures with Trees

Version 1.2.1

Description A set of tools for inspecting and understanding R data structures inspired by str(). Includes ast() for visualizing abstract syntax trees, ref() for showing shared references, cst() for showing call stack trees, and obj_size() for computing object sizes.

License MIT + file LICENSE

URL https://lobstr.r-lib.org/, https://github.com/r-lib/lobstr

BugReports https://github.com/r-lib/lobstr/issues

Depends R (>= 3.6.0)

Imports crayon, methods, prettyunits, rlang (>= 1.0.0)

Suggests covr, pillar, pkgdown, testthat (>= 3.0.0)

LinkingTo cpp11 (>= 0.4.2)

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

Config/build/compilation-database true

NeedsCompilation yes

Author Hadley Wickham [aut, cre],
Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2026-04-04 05:10:39 UTC

Contents

`ast` *Display the abstract syntax tree*

Description

This is a useful alternative to `str()` for expression objects.

Usage

```
ast(x)
```

Arguments

`x`	An expression to display. Input is automatically quoted, use `!!` to unquote if you have already captured an expression object.

See Also

Other object inspectors: `ref()`, `src()`, `sxp()`

Examples

```
# Leaves
ast(1)
ast(x)

# Simple calls
ast(f())
ast(f(x, 1, g(), h(i())))
ast(f()())
ast(f(x)(y))

ast((x + 1))

# Displaying expression already stored in object
```

```
x <- quote(a + b + c)
ast(x)
ast(!!x)

# All operations have this same structure
ast(if (TRUE) 3 else 4)
ast(y <- x * 10)
ast(function(x = 1, y = 2) { x + y } )

# Operator precedence
ast(1 * 2 + 3)
ast(!1 + !1)
```

`cst` *Call stack tree*

Description

Shows the relationship between calls on the stack. This function combines the results of `sys.calls()` and `sys.parents()` yielding a display that shows how frames on the call stack are related.

Usage

```
cst()
```

Examples

```
# If all evaluation is eager, you get a single tree
f <- function() g()
g <- function() h()
h <- function() cst()
f()

# You get multiple trees with delayed evaluation
try(f())

# Pay attention to the first element of each subtree: each
# evaluates the outermost call
f <- function(x) g(x)
g <- function(x) h(x)
h <- function(x) x
try(f(cst()))

# With a little ingenuity you can use it to see how NSE
# functions work in base R
with(mtcars, {cst(); invisible()})
invisible(subset(mtcars, {cst(); cyl == 0}))

# You can also get unusual trees by evaluating in frames
# higher up the call stack
```

```
f <- function() g()
g <- function() h()
h <- function() eval(quote(cst()), parent.frame(2))
f()
```

`mem_used` *How much memory is currently used by R?*

Description

`mem_used()` wraps around `gc()` and returns the exact number of bytes currently used by R. Note that changes will not match up exactly to `obj_size()` as session specific state (e.g. .Last.value) adds minor variations.

Usage

```
mem_used()
```

Examples

```
prev_m <- 0; m <- mem_used(); m - prev_m

x <- 1:1e6
prev_m <- m; m <- mem_used(); m - prev_m
obj_size(x)

rm(x)
prev_m <- m; m <- mem_used(); m - prev_m

prev_m <- m; m <- mem_used(); m - prev_m
```

`obj_addr` *Find memory location of objects and their children.*

Description

`obj_addr()` gives the address of the value that `x` points to; `obj_addrs()` gives the address of the components the list, environment, and character vector `x` point to.

Usage

```
obj_addr(x)

obj_addrs(x)
```

Arguments

`x`	An object

Details

obj_addr() has been written in such away that it avoids taking references to an object.

Examples

```
# R creates copies lazily
x <- 1:10
y <- x
obj_addr(x) == obj_addr(y)

y[1] <- 2L
obj_addr(x) == obj_addr(y)

y <- runif(10)
obj_addr(y)
z <- list(y, y)
obj_addrs(z)

y[2] <- 1.0
obj_addrs(z)
obj_addr(y)

# The address of an object is different every time you create it:
obj_addr(1:10)
obj_addr(1:10)
obj_addr(1:10)
```

---

obj_size *Calculate the size of an object.*

Description

obj_size() computes the size of an object or set of objects; obj_sizes() breaks down the individual contribution of multiple objects to the total size.

Usage

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

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

Arguments

...	Set of objects to compute size.
env	Environment in which to terminate search. This defaults to the current environment so that you don't include the size of objects that are already stored elsewhere. Regardless of the value here, obj_size() never looks past the global or base environments.

Value

An estimate of the size of the object, in bytes.

Compared to `object.size()`

Compared to `object.size()`, `obj_size()`:

- Accounts for all types of shared values, not just strings in the global string pool.
- Includes the size of environments (up to `env`)
- Accurately measures the size of ALTREP objects.

Environments

`obj_size()` attempts to take into account the size of the environments associated with an object. This is particularly important for closures and formulas, since otherwise you may not realise that you've accidentally captured a large object. However, it's easy to over count: you don't want to include the size of every object in every environment leading back to the `emptyenv()`. `obj_size()` takes a heuristic approach: it never counts the size of the global environment, the base environment, the empty environment, or any namespace.

Additionally, the `env` argument allows you to specify another environment at which to stop. This defaults to the environment from which `obj_size()` is called to prevent double-counting of objects created elsewhere.

Examples

```
# obj_size correctly accounts for shared references
x <- runif(1e4)
obj_size(x)

z <- list(a = x, b = x, c = x)
obj_size(z)

# this means that object size is not transitive
obj_size(x)
obj_size(z)
obj_size(x, z)

# use obj_size() to see the unique contribution of each component
obj_sizes(x, z)
obj_sizes(z, x)
obj_sizes(!!!z)

# obj_size() also includes the size of environments
f <- function() {
  x <- 1:1e4
  a ~ b
}
obj_size(f())

#' # In R 3.5 and greater, `:` creates a special "ALTREP" object that only
```

```
# stores the first and last elements. This will make some vectors much
# smaller than you'd otherwise expect
obj_size(1:1e6)
```

`ref` *Display tree of references*

Description

This tree display focusses on the distinction between names and values. For each reference-type object (lists, environments, and optional character vectors), it displays the location of each component. The display shows the connection between shared references using a locally unique id.

Usage

```
ref(..., character = FALSE)
```

Arguments

`...`	One or more objects
`character`	If `TRUE`, show references from character vector in to global string pool

See Also

Other object inspectors: `ast()`, `src()`, `sxp()`

Examples

```
x <- 1:100
ref(x)

y <- list(x, x, x)
ref(y)
ref(x, y)

e <- new.env()
e$e <- e
e$x <- x
e$y <- list(x, e)
ref(e)

# Can also show references to global string pool if requested
ref(c("x", "x", "y"))
ref(c("x", "x", "y"), character = TRUE)
```

`src` *Display tree of source references*

Description

View source reference metadata attached to R objects in a tree structure. Shows source file information, line/column locations, and lines of source code.

Usage

```
src(x, max_depth = 5L, max_length = 100L, ...)
```

Arguments

`x`	An R object with source references. Can be: • A `srcref` object • A list of `srcref` objects • A expression vector with attached source references • An evaluated closure with attached source references • A quoted call with attached source references
`max_depth`	Maximum depth to traverse nested structures (default 5)
`max_length`	Maximum number of srcref nodes to display (default 100)
`...`	Additional arguments passed to `tree()`

Value

Returns a structured list containing the source reference information. Print it to view the formatted tree.

Overview

Source references are made of two kinds of objects:

- `srcref` objects, which contain information about a specific location within the source file, such as the line and column numbers.
- `srcfile` objects, which contain metadata about the source file such as its name, path, and encoding.

Where and when are source references created?:

Ultimately the R parser creates source references. The main two entry points to the parser are:

- The R function `parse()`.
- The frontend hook `ReadConsole`, which powers the console input parser in the R CLI and in IDEs. This C-level parser can also be accessed from C code via `R_ParseVector()`.

In principle, anything that calls `parse()` may create source references, but here are the important direct and indirect callers:

- `source()` and `sys.source()` which parse and evaluate code.
- `loadNamespace()` calls `sys.source()` when loading a *source* package: https://github.com/r-devel/r-svn/blob/acd196be/src/library/base/R/namespace.R#L573.
- `R CMD install` creates a lazy-load database from a source package. The first step is to call `loadNamespace()`: https://github.com/r-devel/r-svn/blob/acd196be/src/library/tools/R/makeLazyLoad.R#L32

By default source references are not created but can be enabled by:

- Passing `keep.source = TRUE` explicitly to `parse()`, `source()`, `sys.source()`, or `loadNamespace()`.
- Setting `options(keep.source = TRUE)`. This affects the default arguments of the aforementioned functions, as well as the console input parser. In interactive sessions, `keep.source` is set to `TRUE` by default: https://github.com/r-devel/r-svn/blob/3a4745af/src/library/profile/Common.R#L26.
- Setting `options(keep.source.pkgs = TRUE)`. This affects loading a package from source, and installing a package from source.

`srcref` **objects:**

`srcref` objects are compact integer vectors describing a character range in a source. It records start/end lines and byte/column positions and, optionally, the parsed-line numbers if `#line` directives were used.

Lengths of 4, 6, or 8 are allowed:

- 4: basic (first_line, first_byte, last_line, last_byte). Byte positions are within the line.
- 6: adds columns in Unicode codepoints (first_col, last_col)
- 8: adds parsed-line numbers (first_parsed, last_parsed)

The "column" information does not represent grapheme clusters, but Unicode codepoints. The column cursor is incremented at every UTF-8 lead byte and there is no support for encodings other than UTF-8.

The srcref columns are right-boundary positions, meaning that for an expression starting at the start of a line, the column will be 1. `wholeSrcref` (see below) on the other hand starts at 0, before the first character. It might also end 1 character after the last srcref column.

They are attached as attributes (e.g. `attr(x, "srcref")` or `attr(x, "wholeSrcref")`), possibly wrapped in a list, to the following objects:

- Expression vectors returned by `parse()` (wrapped in a list)
- Quoted function calls (unwrapped)
- Quoted `{` calls (wrapped in a list). This is crucial for debugging: when R steps through brace lists, the srcref for the current expression is saved to a global variable (`R_Srcref`) so the IDE knows exactly where execution is paused. See: https://github.com/r-devel/r-svn/blob/fa0b47c5/src/main/eval.c#L2986.
- Evaluated closures (unwrapped)

They have a `srcfile` attribute that points to the source file.

Methods:

- `as.character()`: Retrieves relevant source lines from the `srcfile` reference.

`wholeSrcref` *attributes:*

These are `srcref` objects stored in the `wholeSrcref` attributes of:

- Expression vectors returned by `parse()`, which seems to be the intended usage.

- { calls, which seems unintended.

For expression vectors, the wholeSrcref spans from the first position to the last position and represents the entire document. For braces, they span from the first position to the location of the closing brace. There is no way to know the location of the opening brace without reparsing, which seems odd. It's probably an overlook from xxexprlist() calling attachSrcrefs() in https://github.com/r-devel/r-svn/blob/52affc16/src/main/gram.y#L1380. That function is also called at the end of parsing, where it's intended for the wholeSrcref attribute to be attached.
For evaluated closures, the wholeSrcref attribute on the body has the same unreliable start positions as { nodes.

srcfile objects:

srcfile objects are environments representing information about a source file that a source reference points to. They typically refer to a file on disk and store the filename, working directory, a timestamp, and encoding information.

While it is possible to create bare srcfile objects, specialized subclasses are much more common.

srcfile:

A bare srcfile object does not contain any data apart from the file path. It lazily loads lines from the file on disk, without any caching.
Fields common to all srcfile objects:

- filename: The filename of the source file. If relative, the path is resolved against wd.
- wd: The working directory (getwd()) at the time the srcfile was created, generally at the time of parsing).
- timestamp: The timestamp of the source file. Retrieved from filename with file.mtime().
- encoding: The encoding of the source file.
- Enc: The encoding of output lines. Used by getSrcLines(), which calls iconv() when Enc does not match encoding.
- parseData (optional): Parser information saved when keep.source.data is set to TRUE.

Implementations:

- print() and summary() to print information about the source file.
- open() and close() to access the underlying file as a connection.

Helpers:

- getSrcLines(): Retrieves source lines from a srcfile.

srcfilecopy:

A srcfilecopy stores the actual source lines in memory in $lines. srcfilecopy is useful when the original file may change or does not exist, because it preserves the exact text used by the parser.
This type of srcfile is the most common. It's created by:

- The R-level parse() function when text is supplied:

```
# Creates a `"<text>"` non-file `srcfilecopy`
parse(text = "...", keep.source = TRUE)
```

- The console's input parser when getOption("keep.source") is TRUE.
- sys.source() when keep.source = TRUE:

```
sys.source(file, keep.source = TRUE)
```

The srcfilecopy object is timestamped with the file's last modification time. https://github.com/r-devel/r-svn/blob/52affc16/src/library/base/R/source.R#L273-L276

Fields:

- filename: The filename of the source file. If isFile is FALSE, the field is non meaningful. For instance parse(text =) sets it to "<text>", and the console input parser sets it to "".
- isFile: A logical indicating whether the source file exists.
- fixedNewlines: If TRUE, lines is a character vector of lines with no embedded \n characters. The getSrcLines() helper regularises lines in this way and sets fixedNewlines to TRUE.

Note that the C-level parser (used directly mainly when parsing console input) does not call the R-level constructor and only instantiates the filename (set to "") and lines fields.

srcfilealias*:*
This object wraps an existing srcfile object (stored in original). It allows exposing a different filename while delegating the open/close/get lines operations to the srcfile stored in original.
The typical way aliases are created is via #line *line* *filename* directives where the optional *filename* argument is supplied. These directives remap the srcref and srcfile of parsed code to a different location, for example from a temporary file or generated file to the original location on disk.
Created by install.packages() when installing a *source* package with keep.source.pkgs set to TRUE (see https://github.com/r-devel/r-svn/blob/52affc16/src/library/tools/R/install.R#L545), but only when:

- Encoding was supplied in DESCRIPTION
- The system locale is not "C" or "POSIX".

The source files are converted to the encoding of the system locale, then collated in a single source file with #line directives mapping them to their original file names (with full paths): https://github.com/r-devel/r-svn/blob/52affc16/src/library/tools/R/admin.R#L342.
Note that the filename of the original srcfile incorrectly points to the package path in the install destination.
Fields:

- filename: The virtual file name (or full path) of the parsed code.
- original: The actual srcfile the code was parsed from.

See Also

- srcfile(): Base documentation for srcref and srcfile objects.
- getParseData(): Parse information stored when keep.source.data is TRUE.
- Source References (R Journal): https://journal.r-project.org/articles/RJ-2010-010/

Other object inspectors: ast(), ref(), sxp()

sxp *Inspect an object*

Description

`sxp(x)` is similar to `.Internal(inspect(x))`, recursing into the C data structures underlying any R object. The main difference is the output is a little more compact, it recurses fully, and avoids getting stuck in infinite loops by using a depth-first search. It also returns a list that you can compute with, and carefully uses colour to highlight the most important details.

Usage

```
sxp(x, expand = character(), max_depth = 5L)
```

Arguments

`x`	Object to inspect
`expand`	Optionally, expand components of the true that are usually suppressed. Use: • "character" to show underlying entries in the global string pool. • "environment" to show binding components without any side effects (e.g. promises or active bindings). • "altrep" to show the underlying data. • "call" to show the full AST (but `ast()` is usually superior) • "bytecode" to show generated bytecode.
`max_depth`	Maximum depth to recurse. Use `max_depth = Inf` (with care!) to recurse as deeply as possible. Skipped elements will be shown as `....`

Details

The name `sxp` comes from `SEXP`, the name of the C data structure that underlies all R objects.

See Also

Other object inspectors: `ast()`, `ref()`, `src()`

Examples

```
x <- list(
  TRUE,
  1L,
  runif(100),
  "3"
)
sxp(x)

# Expand "character" to see underlying CHARSXP entries in the global
# string pool
```

```
x <- c("banana", "banana", "apple", "banana")
sxp(x)
sxp(x, expand = "character")

# Expand altrep to see underlying data
x <- 1:10
sxp(x)
sxp(x, expand = "altrep")

# Expand environments to see promise expressions without forcing
e <- new.env(parent = emptyenv())
delayedAssign("x", 1 + 1, assign.env = e)

sxp(e)
sxp(e, expand = "environment")
```

`tree` *Pretty tree-like object printing*

Description

A cleaner and easier to read replacement for `str` for nested list-like objects

Usage

```
tree(
  x,
  ...,
  index_unnamed = FALSE,
  max_depth = 10L,
  max_length = 1000L,
  max_vec_len = 10L,
  show_environments = TRUE,
  hide_scalar_types = TRUE,
  val_printer = crayon::blue,
  class_printer = crayon::silver,
  show_attributes = FALSE,
  remove_newlines = TRUE,
  tree_chars = box_chars()
)
```

Arguments

`x`	A tree like object (list, etc.)
`...`	Ignored (used to force use of names)
`index_unnamed`	Should children of containers without names have indices used as stand-in?
`max_depth`	How far down the tree structure should be printed. E.g. `1` means only direct children of the root element will be shown. Useful for very deep lists.

`max_length` How many elements should be printed? This is useful in case you try and print an object with 100,000 items in it.

`max_vec_len` How many elements should be printed for vectors?

`show_environments`
Should environments be treated like normal lists and recursed into?

`hide_scalar_types`
Should atomic scalars be printed with type and length like vectors? E.g. `x <- "a"` would be shown as `x<char [1]>: "a"` instead of `x: "a"`.

`val_printer` Function that values get passed to before being drawn to screen. Can be used to color or generally style output.

`class_printer` Same as `val_printer` but for the the class types of non-atomic tree elements.

`show_attributes`
Should attributes be printed as a child of the list or avoided?

`remove_newlines`
Should character strings with newlines in them have the newlines removed? Not doing so will mess up the vertical flow of the tree but may be desired for some use-cases if newline structure is important to understanding object state.

`tree_chars` List of box characters used to construct tree. Needs elements `$h` for horizontal bar, `$hd` for dotted horizontal bar, `$v` for vertical bar, `$vd` for dotted vertical bar, `$l` for l-bend, and `$j` for junction (or middle child).

Value

console output of structure

Examples

```
x <- list(
  list(id = "a", val = 2),
  list(
    id = "b",
    val = 1,
    children = list(
      list(id = "b1", val = 2.5),
      list(
        id = "b2",
        val = 8,
        children = list(
          list(id = "b21", val = 4)
        )
      )
    )
  ),
  list(
    id = "c",
    val = 8,
    children = list(
      list(id = "c1"),
      list(id = "c2", val = 1)
```

```
    )
  )
)

# Basic usage
tree(x)

# Even cleaner output can be achieved by not printing indices
tree(x, index_unnamed = FALSE)

# Limit depth if object is potentially very large
tree(x, max_depth = 2)

# You can customize how the values and classes are printed if desired
tree(x, val_printer = function(x) {
  paste0("_", x, "_")
})
```

`tree_label` *Build element or node label in tree*

Description

These methods control how the value of a given node is printed. New methods can be added if support is needed for a novel class

Usage

```
tree_label(x, opts)
```

Arguments

`x`	A tree like object (list, etc.)
`opts`	A list of options that directly mirrors the named arguments of tree. E.g. `list(val_printer = crayon::red)` is equivalent to `tree(..., val_printer = crayon::red)`.

Index