Package 'glyrepr'

October 22, 2026

Title Representation for Glycan Compositions and Structures

Version 1.2.0

Description Computational representations of glycan compositions and structures, including details such as linkages, anomers, and substituents. Supports varying levels of monosaccharide specificity (e.g., ``Hex" or ``Gal") and ambiguous linkages. Provides robust parsing and generation of IUPAC-condensed structure strings. Optimized for vectorized operations on glycan structures, with efficient handling of duplications. As the cornerstone of the glycoverse ecosystem, this package delivers the foundational data structures that power glycomics and glycoproteomics analysis workflows.

License MIT + file LICENSE

Suggests testthat (>= 3.0.0), patrick, knitr, rmarkdown, tictoc, lobstr, withr

Config/testthat/edition 3

Encoding UTF-8

URL https://glycoverse.github.io/glyrepr/, https://github.com/glycoverse/glyrepr

Imports Rcpp, checkmate, cli, dplyr (>= 1.2.0), glue, igraph, pillar, purrr, rlang, rstackdeque, stringr, tibble, vctrs (>= 0.6.5)

VignetteBuilder knitr

BugReports https://github.com/glycoverse/glyrepr/issues

RoxygenNote 7.3.3

LinkingTo Rcpp

SystemRequirements C++17

NeedsCompilation yes

Author Bin Fu [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-8567-2997>)

Maintainer Bin Fu <23110220018@m.fudan.edu.cn>

Depends R (>= 4.1.0)

Repository CRAN

Date/Publication 2026-10-22 14:50:02 UTC

Contents

as_glycan_composition . . . 3
as_glycan_structure . . . 4
available_monosaccharides . . . 6
available_substituents . . . 7
canonicalize_glycan_graph . . . 7
canonicalize_glycan_graphs . . . 9
convert_to_generic . . . 10
count_mono . . . 12
enumerate_floating_graph_localizations . . . 13
enumerate_floating_localizations . . . 15
fill_anomer_pos . . . 17
get_alditol . . . 18
get_anomer . . . 18
get_mono_type . . . 19
get_structure_graphs . . . 20
get_structure_level . . . 21
glycan_composition . . . 22
glycan_structure . . . 23
graph_to_iupac . . . 27
has_floating_parts . . . 29
has_floating_substituents . . . 30
has_linkages . . . 31
infer_anomer_pos . . . 32
is_known_monosaccharide . . . 33
localize_floating_parts . . . 33
new_glycan_structure . . . 35
n_glycan_core . . . 37
possible_linkages . . . 38
remove_linkages . . . 39
remove_substituents . . . 40
simap . . . 40
smap . . . 42
smap2 . . . 44
smap_predicates . . . 46
smap_unique . . . 47
spmap . . . 48
structure_candidate_edges . . . 50
structure_component_membership . . . 51
structure_floating_candidates . . . 52
structure_from_arrays . . . 53
structure_tables . . . 54
structure_to_iupac . . . 56
validate_glycan_graph . . . 58
validate_glycan_graph_vector . . . 60
valid_linkages . . . 62

Index **63**

as_glycan_composition *Convert to Glycan Composition*

Description

Converts an object to a glycan composition using vctrs casting framework. This function provides a convenient way to convert various input types to `glycan_composition()`.

Usage

```
as_glycan_composition(x)
```

Arguments

x	An object to convert to a glycan composition. Supported inputs include: • Named integer vectors or lists of named integer vectors • Character vectors with composition strings (e.g., "Hex(5)HexNAc(2)") • `glyrepr_structure` objects (counts both monosaccharides and substituents) • Glycan `igraph` objects (returns a length-one composition vector) • Existing `glyrepr_composition` objects (returned as-is)

Details

This function uses the vctrs casting framework for type conversion. When converting from glycan structures, both monosaccharides and substituents are counted. Substituents are extracted from the `sub` attribute of each vertex and from the `floating_substituents` graph attribute. For example, a vertex with `sub = "3Me"` or an unresolved `{?Me}` each contributes one "Me" substituent to the composition.

Simple composition strings use one-letter residue codes: "H" for "Hex", "N" for "HexNAc", "F" for "dHex", "S"/"A" for "NeuAc", and "G" for "NeuGc". "E" and "L" are also accepted as linkage-specific Neu5Ac codes; they are converted to "NeuAc" with a warning because composition objects do not preserve linkage information.

Value

A `glyrepr_composition` object.

Examples

```
# From a single named vector
as_glycan_composition(c(Hex = 5, HexNAc = 2))

# From a list of named vectors
as_glycan_composition(list(c(Hex = 5, HexNAc = 2), c(Hex = 3, HexNAc = 1)))

# From a character vector of Byonic composition strings
as_glycan_composition(c("Hex(5)HexNAc(2)", "Hex(3)HexNAc(1)"))
```

```
# From a character vector of simple composition strings
as_glycan_composition(c("H5N2", "H5N4S1F1"))

# From an existing composition (returns as-is)
comp <- glycan_composition(c(Hex = 5, HexNAc = 2))
as_glycan_composition(comp)

# From a glycan structure vector or graph
strucs <- c(n_glycan_core(), o_glycan_core_1())
as_glycan_composition(strucs)
graph <- get_structure_graphs(strucs[[1]])
as_glycan_composition(graph)
```

`as_glycan_structure` *Convert to Glycan Structure Vector*

Description

Convert an object to a glycan structure vector.

Usage

```
as_glycan_structure(
  x,
  on_failure = c("error", "na"),
  progress = FALSE,
  warn = TRUE
)
```

Arguments

`x`	An object to convert to a glycan structure vector. Can be an igraph object, a list of igraph objects, a character vector of IUPAC-condensed strings, or an existing glyrepr_structure object.
`on_failure`	The failure policy for element-local parsing, validation, and canonicalization errors. `"error"` preserves the default strict behavior. `"na"` replaces failed elements with `NA` and emits one warning that reports their positions and failure reasons. Existing missing elements remain missing without a warning. Vector-level incompatibilities still produce an error.
`progress`	`FALSE` (default) for silent operation, `TRUE` for a cli progress bar, or a function with arguments `stage`, `current`, and `total`. A callback owns its display; no cli bar is created. Counts are local to each stage, not an overall percentage. Character parsing counts unique non-missing inputs; graph construction may count deduplicated canonical structures. Updates are throttled, with stage boundaries always reported. Empty or already-converted inputs may report no stages. Callback errors abort the operation, including with `on_failure = "na"`. The cli bar

is closed when the call exits, including on errors or interrupts, and follows cli display settings (short or non-interactive calls may show no visible bar).

`warn` Whether to warn about recovered failures with `on_failure = "na"`. Validation is always performed.

Details

Character input assumes the natural absolute configuration for unprefixed monosaccharides. Less common configurations use a leading `D-` or `L-`, such as `D-Fuc`, `L-Gul`, and `D-Fucf`. Alditols use `-ol` on the main reducing-end residue, for example `Gal(b1-4)GlcNAc-ol(a1-`. The reducing-end anomer annotation remains part of the canonical representation.

Character input supports floating-part blocks before the main IUPAC-condensed structure. `{Neu5Ac(a2-3)}<main>` allows every feasible node outside its own component as a candidate parent, while an explicit `|<parents>` suffix restricts that domain. Parent indices follow residue order in the complete supplied sequence: residues in floating blocks are counted left to right before the main glycan, and substituent blocks add no indices. A floating part may target another floating component or the main tree, but cannot target itself. Indices are remapped to canonical complete sequence order in the result. The suffix is a `glyrepr` extension to curly-brace IUPAC notation. A singleton candidate set is accepted as input but fully localizes the attachment, so `{Neu5Ac(a2-3)|2}Gal(b1-4)GlcNAc(b1-` canonicalizes to the ordinary structure `Neu5Ac(a2-3)Gal(b1-4)GlcNAc(b1-`.

Floating substituents use the same leading-brace and candidate-parent syntax. For example, `{6S}<main>` leaves the sulfated residue unrestricted across all residue nodes, `{6S|1,2}<main>` restricts it to complete-sequence nodes 1 and 2, and `{?S}<main>` also leaves the carbon position unknown. A singleton candidate is normalized into the selected residue's ordinary sub attribute.

Value

A glyrepr_structure object.

Examples

```
library(igraph)

# Convert a single igraph
graph <- make_graph(~ 1-+2)
V(graph)$mono <- c("GlcNAc", "GlcNAc")
V(graph)$sub <- ""
E(graph)$linkage <- "b1-4"
graph$anomer <- "a1"
as_glycan_structure(graph)

# Convert a list of igraphs
o_glycan_vec <- o_glycan_core_1()
o_glycan_graph <- get_structure_graphs(o_glycan_vec)
as_glycan_structure(list(graph, o_glycan_graph))

# Convert a character vector of IUPAC-condensed strings
as_glycan_structure(c("GlcNAc(b1-4)GlcNAc(b1-", "Man(a1-2)GlcNAc(b1-"))
as_glycan_structure(c("D-Fuc(a1-", "L-Gul(b1-", "D-Fucf(a1-"))
as_glycan_structure("Gal(b1-4)GlcNAc-ol(a1-")
```

```
# Parse a floating residue with two candidate parents
floating_iupac <- paste0(
  "{Neu5Ac(a2-3)|2,5}",
  "Gal(b1-4)GlcNAc(b1-2)Man(a1-3)",
  "[Gal(b1-4)GlcNAc(b1-2)Man(a1-6)]",
  "Man(b1-4)GlcNAc(b1-4)GlcNAc(b1-"
)
as_glycan_structure(floating_iupac)

# Preserve valid elements while replacing an invalid element with NA
as_glycan_structure(
  c(valid = "Glc(?1-", invalid = "not-a-structure"),
  on_failure = "na"
)
```

`available_monosaccharides` *Get Available Monosaacharides*

Description

This function returns a character vector of monosaccharide names of the given type. See `get_mono_type()` for monosaacharide types. Concrete furanose forms use an `f` after the monosaccharide stem, such as `Galf` and `GlcfNAc`. Generic names do not encode ring form. Less common absolute configurations use a leading `D-` or `L-`, such as `D-Fuc`, `L-Gul`, and `D-Fucf`. Unprefixed names retain their natural configurations.

Usage

```
available_monosaccharides(mono_type = "all")
```

Arguments

`mono_type`	A character string specifying the type of monosaccharides. Can be "all", "generic", or "concrete". Default is "all".

Value

A character vector of monosaccharide names.

Examples

```
available_monosaccharides()
available_monosaccharides("concrete")
```

available_substituents *Available Substituents*

Description

Get the available substituents for monosaccharides.

Usage

```
available_substituents()
```

Value

A character vector.

Examples

```
available_substituents()
```

canonicalize_glycan_graph *Canonicalize a Glycan Graph*

Description

Add a vertex `name` attribute when needed and reorder the vertices and edges of one glycan graph to match its IUPAC-condensed sequence.

Usage

```
canonicalize_glycan_graph(graph)
```

Arguments

`graph`	A single `igraph` glycan graph.

Details

This function assumes that `graph` has already passed `validate_glycan_graph()`. It performs no semantic validation.

Value

A canonicalized `igraph` glycan graph.

Choosing a construction pipeline

Prefer `as_glycan_structure()` for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use `structure_from_arrays()` to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, `canonicalize_glycan_graphs()` validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. `structure_from_tibbles()` converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. `validate_glycan_graph()` validates one graph; `canonicalize_glycan_graph()` assumes a valid graph; `graph_to_iupac()` assumes a valid, canonical graph. `validate_glycan_graph_vector()` checks the graph-list container, not each graph's semantics. `new_glycan_structure()` trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set

`on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

Other low-level glycan structure functions: `graph_to_iupac()`, `new_glycan_structure()`, `validate_glycan_graph()`, `validate_glycan_graph_vector()`

---

canonicalize_glycan_graphs

Validate and Canonicalize a Batch of Glycan Graphs

Description

Return canonical graphs and their IUPAC keys together. For ordinary trees, ordering and key generation share one traversal. Graph, vertex, and edge attributes are retained with their corresponding objects. Unlike structure vector construction, this function does not deduplicate graphs: equal structures can retain different source attributes.

Usage

```
canonicalize_glycan_graphs(
  graphs,
  validate = TRUE,
  on_failure = c("error", "na")
)
```

Arguments

`graphs`	A list of glycan `igraph` objects. `NULL` elements represent missing structures. List names and positions are preserved.
`validate`	Whether to validate each graph before canonicalization. Set to `FALSE` only for graphs already validated with `validate_glycan_graph()`. This skips semantic validation, not canonicalization. Array records should use `structure_from_arrays()`.
`on_failure`	Either `"error"` (default) or `"na"`. With `"na"`, invalid graphs produce a warning and missing output, with details in `reason`.

Details

Strict failures have class `glyrepr_error_structure_failure` with `position`, `input_name`, and `reason` fields. Recovery warnings have class `glyrepr_warning_structure_failure` with `positions` and `reasons` fields.

Value

A list containing aligned, named vectors `iupac`, `status`, and `reason`, and an aligned named list `graphs`. Status is `"ok"`, `"missing"`, or `"invalid"`. Missing and invalid entries have a `NULL` graph and `NA` key. Reasons are `NA` except for invalid entries.

Examples

```
graphs <- as.list(n_glycan_core())
result <- canonicalize_glycan_graphs(graphs)
result$iupac
new_glycan_structure(result$iupac, stats::setNames(result$graphs, result$iupac))
```

---

convert_to_generic *Convert Monosaccharides to Generic Type*

Description

This function converts monosaccharide types of monosaccharide characters, glycan compositions, or glycan structures from concrete to generic type. This is a simplified version that only supports conversion from "concrete" to "generic" monosaccharides.

Usage

```
convert_to_generic(x)

## S3 method for class 'character'
convert_to_generic(x)

## S3 method for class 'glyrepr_structure'
convert_to_generic(x)

## S3 method for class 'igraph'
convert_to_generic(x)

## S3 method for class 'glyrepr_composition'
convert_to_generic(x)
```

Arguments

x	Either of these objects: • A character of monosaccharide; • A glycan composition vector ("glyrepr_composition" object); • A glycan structure vector ("glyrepr_structure" object); • A glycan `igraph`.

Value

A new object of the same class as `x` with monosaccharides converted to generic type. Graph input retains its vertex IDs and order.

Two types of monosaccharides

There are two types of monosaccharides:

- concrete: e.g. "Gal", "GlcNAc", "Glc", "Fuc", etc.
- generic: e.g. "Hex", "HexNAc", "HexA", "HexN", etc. Concrete furanose forms such as "Galf" and "GlcfNAc" convert to the same generic names as their ringless forms: "Hex" and "HexNAc", respectively. Explicit unusual configurations also convert to the same generic names as their natural counterparts. For example, "D-Fuc" and "D-Fucf" both convert to "dHex".

For the full list of monosaccharides, use `available_monosaccharides()`.

Examples

```
# Convert character vectors
convert_to_generic(c("Gal", "GlcNAc", "Galf", "GlcfNAc"))

# Convert glycan compositions
comps <- glycan_composition(
  c(Gal = 5, GlcNAc = 2),
  c(Glc = 5, GalNAc = 4, Fuc = 1)
)
convert_to_generic(comps)

# Convert glycan structures
strucs <- c(n_glycan_core(), o_glycan_core_1())
convert_to_generic(strucs)
```

count_mono *Get the Number of Monosaccharides*

Description

When mono is:

- NULL (default), returns the total number of monosaccharides and substituents.
- A string, returns the number of the specified monosaccharide or substituent.

Usage

```
count_mono(x, mono = NULL, include_subs = FALSE)

## S3 method for class 'glyrepr_composition'
count_mono(x, mono = NULL, include_subs = FALSE)

## S3 method for class 'glyrepr_structure'
count_mono(x, mono = NULL, include_subs = FALSE)

## S3 method for class 'igraph'
count_mono(x, mono = NULL, include_subs = FALSE)
```

Arguments

x	A glycan composition (glyrepr_composition), a glycan structure (glyrepr_structure) vector, or a glycan igraph.
mono	The monosaccharide or substituent to count. A character scalar. If NULL (default), return the total number of monosaccharides.
include_subs	Whether to include substituents when mono is NULL. Default is FALSE.

Details

When mono is "generic" (e.g. "Hex", "HexNAc"), it counts all "concrete" monosaccharides that match. For example, "Hex" will count all Glc, Man, Gal, etc. When mono is "concrete" (e.g. "Gal", "GalNAc"), NA is returned when the composition contains generic residues. Floating substituents are counted like substituents attached to known residues.

Value

A numeric vector of the same length as x, or a numeric scalar for graph input.

Examples

```
comp <- glycan_composition(c(Gal = 1, Man = 1, GalNAc = 1))
count_mono(comp, "Hex")
count_mono(comp, "Gal")

struct <- as_glycan_structure("Gal(b1-3)GlcNAc(b1-4)Glc(a1-")
count_mono(struct, "Gal")

# Total number of monosaccharides
count_mono(comp)
```

---

enumerate_floating_graph_localizations

Enumerate Floating Graph Localizations

Description

`enumerate_floating_graph_localizations()` generates every conflict-free, fully localized graph permitted by the candidate-parent domains in `graph`. It is the graph-level counterpart to `enumerate_floating_localizatio`

The returned graphs are validated but are not canonicalized. Adding the selected attachment edges does not reorder vertices, so every vertex keeps the same integer ID, name, and attributes as in `graph`. Assignment `parent_node` values therefore refer directly to vertex IDs in both the input and localized graphs.

Floating parts are localized as ordinary edges, while floating substituents are localized into the selected parent vertex's `sub` attribute. Every conflict-free acyclic assignment that connects all components to the main tree is retained, even when multiple assignments would produce the same canonical IUPAC-condensed structure. A graph without floating metadata produces one identity row with an empty assignment table.

`max_variants` is a conservative safeguard. It limits the raw Cartesian product before conflict filtering, so the function may ask for a higher bound even when fewer variants would ultimately remain.

Usage

```
enumerate_floating_graph_localizations(graph, max_variants = 256)
```

Arguments

`graph`	A valid glycan `igraph`, optionally containing unresolved floating parts or substituents.
`max_variants`	A positive integer giving the maximum raw candidate combinations allowed.

Value

A tibble with columns:

- `variant_id`: the sequential localization identifier.
- `graph`: a list-column of fully localized `igraph` objects whose vertex IDs are identical to those in `graph`.
- `assignments`: a list-column of tibbles with `glycan_id`, `part_id`, `parent_node`, and `substituent_id`. Exactly one of `part_id` and `substituent_id` is non-missing in each row. `glycan_id` is always `1L`.

Choosing a construction pipeline

Prefer `as_glycan_structure()` for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use `structure_from_arrays()` to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, `canonicalize_glycan_graphs()` validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. `structure_from_tibbles()` converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. `validate_glycan_graph()` validates one graph; `canonicalize_glycan_graph()` assumes a valid graph; `graph_to_iupac()` assumes a valid, canonical graph. `validate_glycan_graph_vector()` checks the graph-list container, not each graph's semantics. `new_glycan_structure()` trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set `on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

`enumerate_floating_localizations()`

Examples

```
glycan <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
graph <- get_structure_graphs(glycan, return_list = FALSE)
localizations <- enumerate_floating_graph_localizations(graph)
localizations$graph
```

---

enumerate_floating_localizations

Enumerate Floating Localizations

Description

`enumerate_floating_localizations()` generates every conflict-free, fully localized structure permitted by the candidate-parent domains in `x`. Each variant records the complete assignment that produced it.

Candidate combinations for floating parts and substituents are validated simultaneously, including linkages or substituents with multiple possible carbon positions. Floating-component dependencies must be acyclic and ultimately connect to the main tree. Variants are canonicalized and, by default, deduplicated by structure. When multiple assignments produce the same canonical structure, the

first assignment in deterministic candidate order is retained. Set `deduplicate = FALSE` to retain every valid assignment and its original-node provenance, including assignments that produce identical canonical structures.

`max_variants` is a conservative per-input safeguard. It limits the raw Cartesian product before conflict filtering or canonical deduplication, so the function may ask for a higher bound even when fewer variants would ultimately remain.

Missing inputs and structures without floating metadata each produce one row with `variant_id = 1L`, the original structure, and an empty assignment table. Empty structure vectors produce a zero-row result.

Usage

```
enumerate_floating_localizations(x, max_variants = 256, deduplicate = TRUE)
```

Arguments

`x`	A glycan structure vector.
`max_variants`	A positive integer giving the maximum raw candidate combinations allowed for each input structure.
`deduplicate`	A logical value. If `TRUE` (default), retain only the first assignment for each canonical structure. If `FALSE`, retain every conflict-free assignment.

Value

A tibble with columns:

- `input_id`: the integer position in `x`.
- `variant_id`: the sequential identifier after optional canonical deduplication.
- `structure`: a `glyrepr_structure` vector column containing fully localized variants.
- `assignments`: a list-column of tibbles with `glycan_id`, `part_id`, `parent_node`, and `substituent_id`. Exactly one of `part_id` and `substituent_id` is non-missing in each row. Here `glycan_id` equals `input_id`.

Examples

```
glycan <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
enumerate_floating_localizations(glycan)
```

`fill_anomer_pos` *Fill Anomer Positions*

Description

Add anomer positions to glycan structures with missing anomer position information. For example, `"Gal(??-?)GalNAc(??-"` is converted to `"Gal(?1-?)GalNAc(?1-"`.

Usage

```
fill_anomer_pos(strucs)
```

Arguments

`strucs`	A `glycan_structure()` vector or glycan `igraph` with concrete or generic monosaccharides.

Details

For anomer positions that are already specified in the input structures, this function does not modify them.

For a structure with floating parts, the reducing-end position is inferred from the root of the main tree. Positions in virtual floating-part attachment linkages are inferred from each floating part's root residue.

Value

An object of the same representation as `strucs` with anomer positions added where missing. Graph input retains its vertex IDs and order.

Examples

```
glycans <- as_glycan_structure(c(
  "Gal(??-?)GalNAc(??-",
  "Neu5Ac(??-?)Gal(??-?)GalNAc(??-"
))
fill_anomer_pos(glycans)
```

get_alditol *Get Alditol Status*

Description

Determine whether the reducing-end residue of a glycan is an alditol. Alditol status is stored as the graph-level logical attribute `alditol`. Legacy graphs without this attribute are treated as non-alditols.

Usage

```
get_alditol(x)
```

Arguments

x	A glycan structure vector or one glycan `igraph`.

Value

A logical vector for structure-vector input, or one logical scalar for graph input. Missing structure-vector elements return `NA`.

Examples

```
glycans <- as_glycan_structure(c(
  reduced = "Gal(b1-4)GlcNAc-ol(a1-",
  ordinary = "Gal(b1-4)GlcNAc(a1-"
))
get_alditol(glycans)
```

get_anomer *Get the Anomeric information*

Description

Get the Anomeric information

Usage

```
get_anomer(x)
```

Arguments

x	A glycan structure vector or a glycan `igraph`.

Value

A character vector for structure-vector input, or a character scalar for graph input.

Examples

```
x <- n_glycan_core()
get_anomer(x)
```

`get_mono_type` *Get Monosaccharide Types*

Description

This function determines the type of monosaccharides in character vectors, glycan compositions, or glycan structures. Supported types are "concrete", "generic", and "mixed" (see details below).

Usage

```
get_mono_type(x)

## S3 method for class 'character'
get_mono_type(x)

## S3 method for class 'glyrepr_structure'
get_mono_type(x)

## S3 method for class 'igraph'
get_mono_type(x)

## S3 method for class 'glyrepr_composition'
get_mono_type(x)
```

Arguments

`x`	Either of these objects: • A character vector of monosaccharide names; • A glycan composition vector ("glyrepr_composition" object); • A glycan structure vector ("glyrepr_structure" object); • A glycan `igraph`.

Value

- For character input, returns a character vector of the same length as `x`.
- For `glyrepr_structure` and `glyrepr_composition` input, returns one value per element. Missing elements return `NA_character_`.
- For `igraph` input, returns a character scalar. Character and structure-vector outputs preserve input names.

Two types of monosaccharides

There are two types of monosaccharides:

- concrete: e.g. "Gal", "GlcNAc", "Glc", "Fuc", etc.
- generic: e.g. "Hex", "HexNAc", "HexA", "HexN", etc.

For the full list of monosaccharides, use `available_monosaccharides()`.

Special monosaccharides

Some monosaccharides are special in that they have no generic names in database or literature. For example, "Mur" is a rare monosaccharide that has no popular generic name. In `glyrepr`, we assign a "g" prefix to these monosaccharides as their generic names. This includes "gNeu", "gKdn", "gPse", "gLeg", "gAci", "g4eLeg", "gBac", "gKdo", "gMur". These names might only be meaningful inside `glycoverse`. Take care when you export results from `glycoverse` functions to other analysis tools.

See Also

`convert_to_generic()`

Examples

```
# Character vector
get_mono_type(c("Gal", "Hex"))

# Glycan structures
get_mono_type(n_glycan_core(mono_type = "concrete"))
get_mono_type(n_glycan_core(mono_type = "generic"))

# Glycan compositions
comp <- glycan_composition(c(Glc = 2, GalNAc = 1))
get_mono_type(comp)
```

`get_structure_graphs` *Access Individual Glycan Structures*

Description

Extract individual glycan structure graphs from a glycan structure vector. A structure with floating parts is returned as one annotated, weakly disconnected `igraph`: its main tree and floating components share the graph, and the `floating_parts` graph attribute records each component's node indices, virtual attachment, and candidate parents. See `glycan_structure()` for the metadata schema. A structure with floating substituents carries a `floating_substituents` graph attribute containing their tokens and candidate parent indices.

Usage

```
get_structure_graphs(x, return_list = NULL)
```

Arguments

x	A glycan structure vector.
return_list	If TRUE, always returns a list. If FALSE and x has a length of 1, return the igraph object directly. If not provided (default), FALSE when x has a length of 1 and TRUE otherwise, including for an empty vector.

Value

A list of igraph objects or an igraph object directly (see return_list parameter).

Examples

```
structures <- c(o_glycan_core_1(), n_glycan_core())
get_structure_graphs(structures)
get_structure_graphs(structures)
```

---

get_structure_level *Get the Structure Resolution Levels*

Description

Glycan structures can have three possible levels of resolution, determined only by linkage and anomer information:

- "intact": No linkage or anomer is unknown or ambiguous.
- "partial": At least one linkage or anomer is unknown or ambiguous, and at least one linkage or anomer contains known information.
- "topological": All linkages and anomers are completely unknown ("??-?"/"??").

Floating metadata does not by itself affect the structure level. A floating part's attachment linkage is evaluated like an ordinary linkage, while candidate-parent ambiguity for floating parts and substituents is independent of linkage resolution. Consequently, a glycan with floating metadata is regarded as "intact" when every graph-edge and floating-part attachment linkage, as well as the reducing-end anomer, is fully specified, even though a parent residue remains unlocalized.

Usage

```
get_structure_level(x)
```

Arguments

x	A glycan_structure() vector or a glycan igraph.

Value

For vector input, a character vector containing one structure level per element. Missing elements return NA_character_, and an empty input returns character(). For graph input, a character scalar.

See Also

has_linkages(), has_floating_parts(), has_floating_substituents(), get_mono_type()

Examples

```
glycan <- as_glycan_structure("Gal(b1-3)GalNAc(a1-")
get_structure_level(glycan)

floating <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
get_structure_level(floating)
```

---

glycan_composition *Create a Glycan Composition*

Description

Create a glycan composition from a list of named integer vectors. Compositions can contain both monosaccharides and substituents.

Usage

```
glycan_composition(...)

is_glycan_composition(x)
```

Arguments

...	Named integer vectors. Names are monosaccharides or substituents, values are numbers of residues. Monosaccharides and substituents can be mixed in the same composition.
x	A list of named integer vectors.

Details

Compositions can contain:

- Monosaccharides: generic (e.g., "Hex", "HexNAc") or concrete (e.g., "Glc", "Gal"). Generic and concrete residues may be mixed.
- Substituents: e.g., "Me", "Ac", "S". These can be mixed with either generic or concrete monosaccharides.

Components are automatically sorted with monosaccharides first (according to their order in the monosaccharides table), followed by substituents (according to their order in available_substituents()). Duplicate components are automatically summed.

Value

A glyrepr_composition object.

See Also

`available_monosaccharides()`, `available_substituents()`

Examples

```
# A vector with one composition (generic monosaccharides)
glycan_composition(c(Hex = 5, HexNAc = 2))
# A vector with multiple compositions
glycan_composition(c(Hex = 5, HexNAc = 2), c(Hex = 5, HexNAc = 4, dHex = 2))
# Residues are reordered automatically
glycan_composition(c(HexNAc = 1, Hex = 2))
# An example for generic monosaccharides
glycan_composition(c(Hex = 2, HexNAc = 1))
# An example for concrete monosaccharides
glycan_composition(c(Glc = 2, Gal = 1))
# Compositions with substituents
glycan_composition(c(Glc = 1, S = 1))
glycan_composition(c(Hex = 3, HexNAc = 2, Me = 1, Ac = 1))
# Substituents are sorted after monosaccharides
glycan_composition(c(S = 1, Gal = 1, Ac = 1, Glc = 1))
```

---

glycan_structure *Create a Glycan Structure Vector*

Description

`glycan_structure()` creates an efficient glycan structure vector for storing and processing glycan molecular structures. The function employs hash-based deduplication mechanisms, making it suitable for glycoproteomics, glycomics analysis, and glycan structure comparison studies.

Usage

```
glycan_structure(...)

is_glycan_structure(x)
```

Arguments

`...`	igraph graph objects to be converted to glycan structures, or existing glycan structure vectors. Supports mixed input of multiple objects.
`x`	An object to check or convert.

Value

A `glyrepr_structure` class glycan structure vector object.

Data Structure Overview

A glycan structure vector is a vctrs vector with an additional S3 class `glyrepr_structure`.

Each glycan structure must satisfy the following constraints:

Graph Structure Requirements:

- An ordinary structure must be a directed outward tree (reducing end as root).
- A structure with floating parts must be one annotated forest containing exactly one main outward tree and one outward tree per floating part.
- Floating substituents add graph metadata but no vertices or edges.
- Must have a graph attribute `anomer` in the format "a1" or "b1"
  - Unknown parts can be represented with "?", e.g., "?1", "a?", "??"
- May have a graph attribute `alditol`, containing one logical value. Missing attributes are treated as `FALSE` and canonicalized explicitly.

Node Attributes:

- `mono`: Monosaccharide names, must be known monosaccharide types
  - Generic names: Hex, HexNAc, dHex, NeuAc, etc.
  - Concrete names: Glc, Gal, Man, GlcNAc, etc.
  - Generic and concrete names may be mixed
  - NA values are not allowed
- `sub`: Substituent information
  - Single substituent format: "xY" (x = position, Y = substituent name), e.g., "2Ac", "3S"
  - Ambiguous substituent positions use slash-separated alternatives, e.g., "4/6S", "3/4/6Ac"
  - Multiple substituents separated by commas and ordered by position, e.g., "3Me,4Ac", "2S,6P"
  - Unknown substituent positions can be repeated, e.g., "?Me,?S"
  - No substituents represented by empty string ""

Edge Attributes:

- `linkage`: Glycosidic linkage information in format "a/bX-Y"
  - Standard format: e.g., "b1-4", "a2-3"
  - Unknown positions allowed: "a1-?", "b?-3", "??-?"
  - Partially unknown positions: "a1-3/6", "a1-3/6/9"
  - NA values are not allowed

Floating Parts:

Floating parts are disconnected substructures whose attachment to the main tree is not fully localized. They are declared by the `floating_parts` graph attribute, a list with one entry per floating component. Each entry contains:

- `root`: the integer vertex index of the floating component root.

- `nodes`: all integer vertex indices in the floating component, ordered as the component appears in the complete IUPAC-condensed sequence.
- `linkage`: the virtual linkage from that root to its unresolved parent.
- `parents`: integer vertex indices outside the floating component. An empty integer vector means that all feasible nodes outside the component are candidates.

Canonical graphs always contain `nodes`. For backward compatibility, input graphs may omit it; `glycan_structure()` derives the component membership before validation and stores `nodes` in the canonical result.

During canonicalization, a floating part with exactly one effective candidate parent is attached to that parent as an ordinary graph edge. Attachments between floating components merge their `nodes` metadata and can resolve further singleton domains. Only unresolved attachments retain floating metadata, where the virtual attachment is metadata rather than an edge and contributes to the canonical structure key.

Floating Substituents:

A floating substituent has known chemistry but an unresolved parent residue. It is declared by the `floating_substituents` graph attribute, a list with one entry per substituent. Each entry contains:

- `substituent`: one canonical substituent token such as `"6S"`, `"4/6Ac"`, or `"?Me"`.
- `parents`: integer residue vertex indices in the complete structure. An empty integer vector means that all feasible residue nodes are candidates.

A singleton candidate is normalized into the corresponding vertex's `sub` attribute. Candidate parents must permit a conflict-free assignment of occupied carbon positions. Floating-part assignments must also be acyclic and connect every floating component to the main tree.

Node and Edge Order

For an ordinary tree, the indices of vertices and linkages correspond directly to their order in the printed IUPAC-condensed string. For example, for the glycan `Man(a1-3)[Man(a1-6)]Man(b1-4)GlcNAc(b1-4)GlcNAc(b1`
the vertices are "Man", "Man", "Man", "GlcNAc", "GlcNAc", and the linkages are "a1-3", "a1-6", "b1-4", "b1-4".

For a floating structure, floating-component vertices and edges precede the main tree, exactly as their brace-enclosed components precede the main glycan in the complete IUPAC-condensed string. Parent indices written inside braces and stored in `floating_parts$parents` or `floating_substituents$parents` use this same global order. Substituent blocks contribute no vertex indices. A virtual floating attachment is not an edge, and a floating substituent is not a vertex.

NA Support

Glycan structure vectors support NA values for representing missing or unknown structures:

- Create with `glycan_structure(NA)` or `glycan_structure(NULL)`
- Combine with valid structures: `c(struct1, NA, struct2)`
- Convert from character: `as_glycan_structure(c("Glc(a1-", NA))`
- `smap` functions skip NA elements gracefully
- `is.na()` returns `TRUE` for NA elements

Naming Support

Glycan structure vectors can have names, which are preserved during operations. This is particularly useful when working with the `glymotif` package.

Character conversion

A glycan structure vector is not a character vector. Use `as.character()` to explicitly convert it to IUPAC-condensed strings when needed.

Examples

```
library(igraph)

# Example 1: Create a simple glycan structure GlcNAc(b1-4)GlcNAc
graph <- make_graph(~ 1-+2)  # Create graph with two monosaccharides
V(graph)$mono <- c("GlcNAc", "GlcNAc")  # Set monosaccharide types
V(graph)$sub <- ""  # No substituents
E(graph)$linkage <- "b1-4"  # b1-4 glycosidic linkage
graph$anomer <- "a1"  # a anomeric carbon

# Create glycan structure vector
simple_struct <- glycan_structure(graph)
print(simple_struct)

# Example 2: Use predefined glycan core structures
n_core <- n_glycan_core()  # N-glycan core structure
o_core1 <- o_glycan_core_1()  # O-glycan Core 1 structure

# Example 3: Create complex structure with substituents
complex_graph <- make_graph(~ 1-+2-+3)
V(complex_graph)$mono <- c("GlcNAc", "Gal", "Neu5Ac")
V(complex_graph)$sub <- c("", "", "")  # Add substituents as needed
E(complex_graph)$linkage <- c("b1-4", "a2-3")
complex_graph$anomer <- "b1"

complex_struct <- glycan_structure(complex_graph)
print(complex_struct)

# Example 4: Parse a floating part with explicit candidate parents
floating <- as_glycan_structure(
  "{Neu5Ac(a2-3)|2,3}Gal(b1-3)[Gal(b1-4)]GlcNAc(a1-"
)
structure_floating_parts(floating)

# Example 5: Parse a substituent with two candidate residues
floating_sub <- as_glycan_structure(
  "{6S|1,2}Gal(a1-3)Glc(a1-3)Man(a1-"
)
get_structure_graphs(floating_sub)$floating_substituents

# Example 6: Check if object is a glycan structure
is_glycan_structure(simple_struct)  # TRUE
```

```
is_glycan_structure(graph)          # FALSE
```

---

graph_to_iupac *Generate IUPAC-Condensed from a Glycan Graph*

Description

Generate one IUPAC-condensed string directly from one glycan graph.

Usage

```
graph_to_iupac(graph)
```

Arguments

graph	A single igraph glycan graph.

Details

This low-level function assumes that graph is valid and canonical. It performs no semantic validation or canonicalization.

Value

A single, unnamed IUPAC-condensed string.

Choosing a construction pipeline

Prefer as_glycan_structure() for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use structure_from_arrays() to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, canonicalize_glycan_graphs() validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. structure_from_tibbles() converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. validate_glycan_graph() validates one graph; canonicalize_glycan_graph() assumes a valid graph; graph_to_iupac() assumes a valid, canonical graph. validate_glycan_graph_vector() checks the graph-list container, not each graph's semantics. new_glycan_structure() trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set `on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

Other low-level glycan structure functions: `canonicalize_glycan_graph()`, `new_glycan_structure()`, `validate_glycan_graph()`, `validate_glycan_graph_vector()`

has_floating_parts *Detect Floating Glycan Parts*

Description

Test whether each glycan structure contains one or more unresolved floating parts.

Usage

```
has_floating_parts(x)
```

Arguments

x	A glycan_structure() vector or a glycan igraph.

Details

A floating part is a known glycan residue or substructure whose parent residue in the complete glycan structure is not fully localized. For example, a bi-antennary N-glycan may contain one sialic acid while the available evidence cannot determine which of its two terminal galactoses carries that residue. The sialic acid can then be represented as a floating part with both galactoses as candidate parents.

In the glyrepr IUPAC extension, floating parts appear in braces before the main glycan:

- {<floating>}<main> means every feasible node outside that floating component is a candidate parent.
- {<floating>|<parents>}<main> restricts the candidates to the comma-separated complete-sequence node indices in <parents>.

Node indices follow residue order in the complete IUPAC-condensed sequence: residue nodes in floating blocks are counted from left to right before the main glycan, while substituent blocks contribute no node indices. A floating part may target a node in another floating component or the main tree, but not a node in its own component. It may contain one residue or an entire subtree, and its virtual attachment linkage may be fully known, partially known, or unknown.

Internally, an unresolved structure is an annotated forest containing one main tree and one disconnected tree per floating part. The virtual linkage and candidate parents are stored as graph metadata rather than as an edge. Each floating-part metadata entry also stores nodes, the complete vector of vertex indices in that floating component, so downstream graph operations do not need to rediscover its membership by traversal. structure_floating_parts() exposes attachment metadata in tabular form, and structure_component_membership() exposes component membership.

An effective singleton parent domain fully localizes the attachment. When one floating component attaches to another, their component metadata is merged and newly singleton domains are resolved iteratively. Such parts are normalized to ordinary graph edges, so has_floating_parts() returns FALSE once every attachment is localized.

Value

A logical vector with the same length and names as vector input, or a logical scalar for graph input. Missing structures produce NA.

See Also

structure_floating_parts(), has_floating_substituents(), as_glycan_structure()

Examples

```
main <- paste0(
  "Gal(b1-4)GlcNAc(b1-2)Man(a1-3)",
  "[Gal(b1-4)GlcNAc(b1-2)Man(a1-6)]",
  "Man(b1-4)GlcNAc(b1-4)GlcNAc(b1-"
)
ambiguous <- as_glycan_structure(
  paste0("{Neu5Ac(a2-3)|2,5}", main)
)
glycans <- c(ambiguous = ambiguous, ordinary = as_glycan_structure(main))
has_floating_parts(glycans)
structure_floating_parts(ambiguous)

localized <- as_glycan_structure(
  "{Neu5Ac(a2-3)|2}Gal(b1-4)GlcNAc(b1-"
)
has_floating_parts(localized)
```

---

has_floating_substituents

Detect Floating Substituents

Description

Test whether each glycan structure contains one or more substituents whose parent residue is unresolved.

Usage

```
has_floating_substituents(x)
```

Arguments

x	A glycan_structure() vector or a glycan igraph.

Details

In the glyrepr IUPAC extension, floating substituents appear in braces before the main glycan. {6S}<main> allows every feasible residue in the complete structure to carry the substituent, {6S|1,2}<main> restricts it to complete-sequence nodes 1 and 2, and {?S}<main> also leaves the carbon position unknown. Residue nodes in floating blocks are counted before the main glycan; substituent blocks contribute no node indices.

Internally, unresolved substituents are stored in the floating_substituents graph attribute. It is a list with one entry per substituent. Each entry contains a canonical substituent token and an integer parents vector of candidate graph vertex indices. An empty vector means every feasible residue vertex is a candidate.

A singleton candidate set is normalized into that vertex's ordinary sub attribute. This also happens whenever chemistry leaves only one feasible residue in the complete structure. structure_floating_substituents() exposes the metadata as a normalized table.

Value

A logical vector with the same length and names as vector input, or a logical scalar for graph input. Missing structures produce NA.

See Also

structure_floating_substituents(), has_floating_parts(), as_glycan_structure()

Examples

```
glycans <- as_glycan_structure(c(
  floating = "{6S}Gal(a1-3)Gal(a1-",
  restricted = "{6S|1,2}Gal(a1-3)Glc(a1-3)Man(a1-",
  ordinary = "Gal6S(a1-"
))
has_floating_substituents(glycans)
```

---

has_linkages *Determine if a Glycan Structure has Linkages*

Description

Unknown linkages in a glycan structure are represented by "??-?". Also, a linkage can be partially known (e.g. "a?-?"). This function checks if a glycan structure has linkages, in a strict or lenient way.

Usage

```
has_linkages(glycan, strict = FALSE)
```

Arguments

glycan	A glycan_structure() vector or a glycan igraph.
strict	A logical value. • If FALSE (default), a glycan is considered to have linkages if any linkage is partially known (not "??-?"). • If TRUE, a glycan is considered to have linkages only if all linkages are fully determined (no "?" or multiple positions in the linkage). Linkages include both graph edges and the virtual attachment linkage of each floating part.

Value

A logical vector for structure-vector input, or a logical scalar for graph input.

See Also

remove_linkages(), possible_linkages()

Examples

```
glycan <- o_glycan_core_1(linkage = TRUE)
has_linkages(glycan)
print(glycan)

glycan <- remove_linkages(glycan)
has_linkages(glycan)
print(glycan)

glycan <- as_glycan_structure("Gal(b1-?)GalNAc(a1-")
has_linkages(glycan)
has_linkages(glycan, strict = TRUE)
```

---

infer_anomer_pos *Infer Anomer Positions*

Description

This function infers the anomer position for concrete or generic monosaccharide names.

Usage

```
infer_anomer_pos(mono)

get_anomer_pos(mono)
```

Arguments

mono	A character vector of monosaccharide names.

Value

An integer vector of anomer positions.

Examples

```
infer_anomer_pos(c("Gal", "Hex", "Neu5Ac"))
```

`is_known_monosaccharide` *Check if a Monosaccharide is Known*

Description

This function checks if a vector of monosaccharide names are known.

Usage

```
is_known_monosaccharide(mono)
```

Arguments

`mono`	A character vector of monosaccharide names.

Value

A logical vector.

Examples

```
is_known_monosaccharide(c("Gal", "Hex"))
is_known_monosaccharide(c("X", "Hx", "Nac"))
```

`localize_floating_parts` *Localize Floating Glycan Parts*

Description

`localize_floating_parts()` attaches selected floating parts to caller-supplied parent nodes. The assignments are interpreted against the floating-part rows and canonical node identifiers returned by `structure_floating_candidates()`.

Each selected parent must belong to the floating part's declared candidate domain. For an unrestricted `{<floating>}` part, that domain contains every feasible node outside its own component, including nodes in other floating components. Assignments must also be simultaneously compatible with occupied and potential acceptor linkage positions, acyclic, and ultimately connected to the main tree.

Selected virtual attachments become ordinary graph edges. Unassigned floating parts remain floating. The resulting structures are canonicalized, and candidate-parent indices for remaining parts are remapped to the new canonical complete-sequence order. Attaching one floating component to another merges their component metadata and can iteratively resolve newly singleton domains.

Missing values, vector positions, and names in structure-vector input are preserved. For graph input, `glycan_id` must be `1L`, and selected edges are appended without canonicalizing or renumbering vertices.

Usage

```
localize_floating_parts(x, assignments)
```

Arguments

`x`	A glycan structure vector or a glycan `igraph`.
`assignments`	A data frame with integer columns `glycan_id`, `part_id`, and `parent_node`. Each `(glycan_id, part_id)` pair may occur at most once.

Value

An object of the same representation as `x`. Structure-vector output has the same length and names as `x`; graph output retains its vertex IDs and order.

Examples

```
glycan <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
assignments <- tibble::tibble(
  glycan_id = 1L,
  part_id = 1L,
  parent_node = 2L
)
localize_floating_parts(glycan, assignments)
```

`new_glycan_structure` *Construct a Glycan Structure Vector from Trusted Data*

Description

Assemble a glycan structure vector from IUPAC-condensed values and a graph lookup table without graph validation, canonicalization, IUPAC generation, vector-level compatibility checks, or graph deduplication.

Usage

```
new_glycan_structure(iupac = character(), graphs = list())
```

Arguments

`iupac`	A character vector of canonical IUPAC-condensed strings. Missing values are allowed, and names are preserved exactly.
`graphs`	A named list of valid, canonical `igraph` glycan graphs keyed by IUPAC-condensed strings.

Details

`graphs` must be a named list containing one graph for every distinct, non-missing value in `iupac`. Additional named graphs are allowed so that vctrs prototypes can retain their graph lookup tables. Graph names must be unique and non-missing. This function checks these inexpensive representation invariants but trusts that each graph matches its key.

Value

A `glyrepr_structure` vector.

Choosing a construction pipeline

Prefer `as_glycan_structure()` for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use `structure_from_arrays()` to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, `canonicalize_glycan_graphs()` validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. `structure_from_tibbles()` converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. `validate_glycan_graph()` validates one graph; `canonicalize_glycan_graph()` assumes a valid graph; `graph_to_iupac()` assumes a valid, canonical graph. `validate_glycan_graph_vector()` checks the graph-list container, not each graph's semantics. `new_glycan_structure()` trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set `on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

Other low-level glycan structure functions: `canonicalize_glycan_graph()`, `graph_to_iupac()`, `validate_glycan_graph()`, `validate_glycan_graph_vector()`

`n_glycan_core` *Example Glycan Structures*

Description

Create example glycan structures for testing and demonstration. Includes **N-glycan core** and **O-glycan core 1** and **core 2**.

Usage

```
n_glycan_core(linkage = TRUE, mono_type = "concrete")

o_glycan_core_1(linkage = TRUE, mono_type = "concrete")

o_glycan_core_2(linkage = TRUE, mono_type = "concrete")
```

Arguments

`linkage`	A logical indicating whether to include linkages (e.g. "b1-4"). Default is `TRUE`.
`mono_type`	A character string specifying the type of monosaccharides. Can be "generic" (Hex, HexNAc, dHex, NeuAc, etc.) or "concrete" (Man, Gal, HexNAc, Fuc, etc.). Default is "concrete".

Value

A `glycan_structure()` vector of length one.

N-Glycan Core

N-Glycans are branched oligosaccharides that are bound, most commonly, via GlcNAc to an Asn residue of the protein backbone. A common motif of all N-glycans is the **chitobiose core**, composed of three mannose and two GlcNAc moieties, which is commonly attached to the protein backbone via GlcNAc. The mannose residue is branched and connected via a1,3- and a1,6-glycosidic linkages to the two other mannose building blocks.

```
  Man
a1-6 \   b1-4      b1-4      b1-
      Man -- GlcNAc -- GlcNAc -
a1-3 /
  Man
```

O-Glycan Core

O-Glycans are highly abundant in extracellular proteins. Generally, O-glycans are extended following four major core structures: **core 1**, **core 2**, core 3, and core 4. The first two are by far the most common core structures in O-glycosylation and are found throughout the body.

core 1:

```
          a1-
    GalNAc -
   / b1-3
Gal
```

core 2:

```
GlcNAc
      \ b1-6 a1-
       GalNAc -
      / b1-3
   Gal
```

Examples

```
print(n_glycan_core(), verbose = TRUE)
print(o_glycan_core_1(), verbose = TRUE)
```

`possible_linkages` *Generate Possible Linkages*

Description

Given an obscure linkage format (having "?", e.g. "a2-?"), this function generates all possible linkages based on the format. See `valid_linkages()` for details.

The ranges of possible anomers, first positions, and second positions can be specified using `anomer_range`, `pos1_range`, and `pos2_range`.

Usage

```
possible_linkages(
  linkage,
  anomer_range = c("a", "b"),
  pos1_range = 1:2,
  pos2_range = 1:9,
  include_unknown = FALSE
)
```

Arguments

`linkage`	A linkage string.
`anomer_range`	A character vector of possible anomers. Default is `c("a", "b")`.
`pos1_range`	A numeric vector of possible first positions. Default is `1:2`.
`pos2_range`	A numeric vector of possible second positions. Default is `1:9`.
`include_unknown`	A logical value. If `TRUE`, "?" will be included. Default is `FALSE`.

Value

A character vector of possible linkages.

See Also

has_linkages(), remove_linkages(), valid_linkages()

Examples

```
possible_linkages("a2-?")
possible_linkages("??-2")
possible_linkages("a1-3")
possible_linkages("a?-?", pos1_range = 2, pos2_range = c(2, 3))
possible_linkages("?1-6", include_unknown = TRUE)
```

---

remove_linkages *Remove All Linkages from a Glycan*

Description

This function replaces all graph-edge and floating-part attachment linkages in a glycan structure with "??-?", as well as the reducing end anomer with "??-".

Usage

```
remove_linkages(glycan)
```

Arguments

glycan	A glyrepr_structure vector or a glycan igraph.

Value

An object of the same representation as glycan with all linkages removed. Graph input retains its vertex IDs and order.

Examples

```
glycan <- o_glycan_core_1(linkage = TRUE)
glycan
remove_linkages(glycan)
```

remove_substituents *Remove All Substituents from a Glycan*

Description

This function replaces all vertex substituents in a glycan structure with empty strings and removes unresolved floating substituents.

Usage

```
remove_substituents(glycan)
```

Arguments

glycan	A glyrepr_structure vector or a glycan igraph.

Value

An object of the same representation as glycan with all substituents removed. Graph input retains its vertex IDs and order.

Examples

```
(glycan <- o_glycan_core_1())
remove_substituents(glycan)
```

simap *Map Functions Over Glycan Structure Vectors with Indices*

Description

These functions apply a function to each unique structure in a glycan structure vector along with their corresponding indices, taking advantage of hash-based deduplication to avoid redundant computation. Similar to purrr imap functions, but optimized for glycan structure vectors.

Usage

```
simap(.x, .f, ...)

simap_vec(.x, .f, ..., .ptype = NULL)

simap_lgl(.x, .f, ...)

simap_int(.x, .f, ...)
```

```
simap_dbl(.x, .f, ...)

simap_chr(.x, .f, ...)

simap_structure(.x, .f, ...)
```

Arguments

.x	A glycan structure vector (glyrepr_structure).
.f	A function that takes an igraph object (from .x) and an index/name, returning a result. Can be a function, purrr-style lambda (~ paste(.x, .y)), or a character string naming a function. A structure with floating metadata is passed as one annotated graph.
...	Additional arguments passed to .f.
.ptype	A prototype for the return type (for simap_vec).

Details

These functions only compute .f once for each unique combination of structure and corresponding index/name, then map the results back to the original vector positions. This is much more efficient than applying .f to each element individually when there are duplicate structures.

simap_structure() reuses unchanged graphs and validates and canonicalizes changed graphs returned by .f. A callback that changes vertex identities or components of a floating structure must also update its floating_parts and floating_substituents metadata.

IMPORTANT PERFORMANCE NOTE: Due to the inclusion of position indices, simap functions have **O(total_structures)** time complexity because each position creates a unique combination, even with identical structures.

Alternative: Consider smap() functions if position information is not required.

The index passed to .f is the position in the original vector (1-based). If the vector has names, the names are passed instead of indices.

Return Types:

- simap(): Returns a list with the same length as .x
- simap_vec(): Returns an atomic vector with the same length as .x
- simap_lgl(): Returns a logical vector
- simap_int(): Returns an integer vector
- simap_dbl(): Returns a double vector
- simap_chr(): Returns a character vector
- simap_structure(): Returns a new glycan structure vector (.f must return igraph objects)

Value

- simap(): A list
- simap_vec(): An atomic vector of type specified by .ptype
- simap_lgl(): Returns a logical vector

- `simap_int()`: Returns an integer vector
- `simap_dbl()`: Returns a double vector
- `simap_chr()`: Returns a character vector
- `simap_structure()`: A new glyrepr_structure object

Examples

```
# Create structure vectors with duplicates
core1 <- o_glycan_core_1()
core2 <- n_glycan_core()
structures <- c(core1, core2, core1)  # core1 appears twice

# Map a function that uses both structure and index
simap_chr(structures, function(g, i) paste0("Structure_", i, "_vcount_", igraph::vcount(g)))

# Use purrr-style lambda functions
simap_chr(structures, ~ paste0("Pos", .y, "_vertices", igraph::vcount(.x)))
```

`smap` *Map Functions Over Glycan Structure Vectors*

Description

These functions apply a function to each unique structure in a glycan structure vector, taking advantage of hash-based deduplication to avoid redundant computation. Similar to purrr mapping functions, but optimized for glycan structure vectors.

Usage

```
smap(.x, .f, ...)

smap_vec(.x, .f, ..., .ptype = NULL)

smap_lgl(.x, .f, ...)

smap_int(.x, .f, ...)

smap_dbl(.x, .f, ...)

smap_chr(.x, .f, ...)

smap_structure(.x, .f, ...)
```

Arguments

.x	A glycan structure vector (glyrepr_structure).
.f	A function that takes an igraph object and returns a result. Can be a function, purrr-style lambda (~ .x$attr), or a character string naming a function. A structure with floating metadata is passed as one annotated graph, including its floating_parts and/or floating_substituents graph attributes.
...	Additional arguments passed to .f.
.ptype	A prototype for the return type (for smap_vec).

Details

These functions only compute .f once for each unique structure, then map the results back to the original vector positions. This is much more efficient than applying .f to each element individually when there are duplicate structures.

Structure-returning variants reuse unchanged graphs and validate and canonicalize changed graphs returned by .f. A callback that changes vertex identities or components of a floating structure must also update its floating_parts and floating_substituents metadata.

Return Types:

- smap(): Returns a list with the same length as .x
- smap_vec(): Returns an atomic vector with the same length as .x
- smap_lgl(): Returns a logical vector
- smap_int(): Returns an integer vector
- smap_dbl(): Returns a double vector
- smap_chr(): Returns a character vector
- smap_structure(): Returns a new glycan structure vector (.f must return igraph objects)

Value

- smap(): A list
- smap_vec(): An atomic vector of type specified by .ptype
- smap_lgl/int/dbl/chr(): Atomic vectors of the corresponding type
- smap_structure(): A new glyrepr_structure object

Examples

```
# Create a structure vector with duplicates
core1 <- o_glycan_core_1()
core2 <- n_glycan_core()
structures <- c(core1, core2, core1)  # core1 appears twice

# Map a function that counts vertices - only computed twice, not three times
smap_int(structures, igraph::vcount)

# Map a function that returns logical
```

```
smap_lgl(structures, function(g) igraph::vcount(g) > 5)

# Use purrr-style lambda functions
smap_int(structures, ~ igraph::vcount(.x))
smap_lgl(structures, ~ igraph::vcount(.x) > 5)

# Map a function that modifies structure (must return igraph)
add_vertex_names <- function(g) {
  if (!("name" %in% igraph::vertex_attr_names(g))) {
    igraph::set_vertex_attr(g, "name", value = paste0("v", seq_len(igraph::vcount(g))))
  } else {
    g
  }
}
smap_structure(structures, add_vertex_names)
```

---

smap2 *Map Functions Over Two Glycan Structure Vectors*

Description

These functions apply a function to each unique structure combination in two glycan structure vectors, taking advantage of hash-based deduplication to avoid redundant computation. Similar to purrr map2 functions, but optimized for glycan structure vectors.

Usage

```
smap2(.x, .y, .f, ...)

smap2_vec(.x, .y, .f, ..., .ptype = NULL)

smap2_lgl(.x, .y, .f, ...)

smap2_int(.x, .y, .f, ...)

smap2_dbl(.x, .y, .f, ...)

smap2_chr(.x, .y, .f, ...)

smap2_structure(.x, .y, .f, ...)
```

Arguments

.x	A glycan structure vector (glyrepr_structure).
.y	A vector of the same length as .x, or length 1 (will be recycled).

.f	A function that takes an igraph object (from .x) and a value (from .y) and returns a result. Can be a function, purrr-style lambda (~ .x + .y), or a character string naming a function. A structure with floating metadata is passed as one annotated graph.
...	Additional arguments passed to .f.
.ptype	A prototype for the return type (for smap2_vec).

Details

These functions only compute .f once for each unique combination of structure and corresponding .y value, then map the results back to the original vector positions. This is much more efficient than applying .f to each element pair individually when there are duplicate structure-value combinations.

smap2_structure() reuses unchanged graphs and validates and canonicalizes changed graphs returned by .f. A callback that changes vertex identities or components of a floating structure must also update its floating_parts and floating_substituents metadata.

NA Handling: NA elements in .x are preserved in the output - the function is not applied to NA positions, and the corresponding results are set to NA.

Return Types:

- smap2(): Returns a list with the same length as .x
- smap2_vec(): Returns an atomic vector with the same length as .x
- smap2_lgl(): Returns a logical vector
- smap2_int(): Returns an integer vector
- smap2_dbl(): Returns a double vector
- smap2_chr(): Returns a character vector
- smap2_structure(): Returns a new glycan structure vector (.f must return igraph objects)

Value

- smap2(): A list
- smap2_vec(): An atomic vector of type specified by .ptype
- smap2_lgl/int/dbl/chr(): Atomic vectors of the corresponding type
- smap2_structure(): A new glyrepr_structure object

Examples

```
# Create structure vectors with duplicates
core1 <- o_glycan_core_1()
core2 <- n_glycan_core()
structures <- c(core1, core2, core1)  # core1 appears twice
weights <- c(1.0, 2.0, 1.0)  # corresponding weights

# Map a function that uses both structure and weight
smap2_dbl(structures, weights, function(g, w) igraph::vcount(g) * w)
```

```
# Use purrr-style lambda functions
smap2_dbl(structures, weights, ~ igraph::vcount(.x) * .y)

# Test with recycling (single weight for all structures)
smap2_dbl(structures, 2.5, ~ igraph::vcount(.x) * .y)

# Map a function that modifies structure based on second argument
# This example adds a graph attribute instead of modifying topology
add_weight_attr <- function(g, weight) {
  igraph::set_graph_attr(g, "weight", weight)
}
weights_to_add <- c(1.5, 2.5, 1.5)
smap2_structure(structures, weights_to_add, add_weight_attr)
```

`smap_predicates` *Test Predicates on Glycan Structure Vectors*

Description

These functions test predicates on unique structures in a glycan structure vector, taking advantage of hash-based deduplication to avoid redundant computation. Similar to purrr predicate functions, but optimized for glycan structure vectors.

Usage

```
ssome(.x, .p, ...)

severy(.x, .p, ...)

snone(.x, .p, ...)
```

Arguments

`.x`	A glycan structure vector (glyrepr_structure).
`.p`	A predicate function that takes an igraph object and returns a logical value. Can be a function, purrr-style lambda (`~ .x$attr`), or a character string naming a function.
`...`	Additional arguments passed to `.p`.

Details

These functions only evaluate `.p` once for each unique structure, making them much more efficient than applying `.p` to each element individually when there are duplicate structures.

Return Values:

- `ssome()`: Returns `TRUE` if at least one unique structure satisfies the predicate
- `severy()`: Returns `TRUE` if all unique structures satisfy the predicate
- `snone()`: Returns `TRUE` if no unique structures satisfy the predicate

Value

A single logical value.

Examples

```
# Create a structure vector with duplicates
core1 <- o_glycan_core_1()
core2 <- n_glycan_core()
structures <- c(core1, core2, core1)  # core1 appears twice

# Test if some structures have more than 5 vertices
ssome(structures, function(g) igraph::vcount(g) > 5)

# Test if all structures have at least 3 vertices
severy(structures, function(g) igraph::vcount(g) >= 3)

# Test if no structures have more than 20 vertices
snone(structures, function(g) igraph::vcount(g) > 20)

# Use purrr-style lambda functions
ssome(structures, ~ igraph::vcount(.x) > 5)
severy(structures, ~ igraph::vcount(.x) >= 3)
snone(structures, ~ igraph::vcount(.x) > 20)
```

---

smap_unique *Apply Function to Unique Structures Only*

Description

Apply a function only to the unique structures in a glycan structure vector, returning results in the same order as the unique structures appear. This is useful when you need to perform expensive computations but only care about unique results.

Usage

```
smap_unique(.x, .f, ...)
```

Arguments

`.x`	A glycan structure vector (glyrepr_structure).
`.f`	A function that takes an igraph object and returns a result. Can be a function, purrr-style lambda (`~ .x$attr`), or a character string naming a function. A structure with floating metadata is passed as one annotated graph.
`...`	Additional arguments passed to `.f`.

Value

A list with results for each unique structure, named by their hash codes.

Examples

```
# Create a structure vector with duplicates
core1 <- o_glycan_core_1()
structures <- c(core1, core1, core1)  # same structure 3 times

# Only compute once for the unique structure
unique_results <- smap_unique(structures, igraph::vcount)
length(unique_results)  # 1, not 3

# Use purrr-style lambda
unique_results2 <- smap_unique(structures, ~ igraph::vcount(.x))
length(unique_results2)  # 1, not 3
```

---

spmap *Map Functions Over Glycan Structure Vectors and Multiple Arguments*

Description

These functions apply a function to each unique structure in a glycan structure vector along with corresponding elements from multiple other vectors, taking advantage of hash-based deduplication to avoid redundant computation. Similar to purrr pmap functions, but optimized for glycan structure vectors.

Usage

```
spmap(.l, .f, ...)

spmap_vec(.l, .f, ..., .ptype = NULL)

spmap_lgl(.l, .f, ...)

spmap_int(.l, .f, ...)

spmap_dbl(.l, .f, ...)

spmap_chr(.l, .f, ...)

spmap_structure(.l, .f, ...)
```

Arguments

`.l`	A list where the first element is a glycan structure vector (glyrepr_structure) and the remaining elements are vectors of the same length or length 1 (will be recycled).

.f	A function that takes an igraph object (from first element of .l) and values from other elements, returning a result. Can be a function, purrr-style lambda (~ .x + .y + .z), or a character string naming a function. A structure with floating metadata is passed as one annotated graph.
...	Additional arguments passed to .f.
.ptype	A prototype for the return type (for spmap_vec).

Details

These functions only compute .f once for each unique combination of structure and corresponding values from other vectors, then map the results back to the original vector positions.

spmap_structure() reuses unchanged graphs and validates and canonicalizes changed graphs returned by .f. A callback that changes vertex identities or components of a floating structure must also update its floating_parts and floating_substituents metadata.

NA Handling: NA elements in the first argument (glycan structure vector) are preserved in the output.

Time Complexity Performance:

Performance scales with unique combinations of all arguments rather than total vector length. When argument vectors are highly redundant, performance approaches O(unique_structures). Scaling factor shows time increase when vector size increases 20x.

Return Types:

- spmap(): Returns a list with the same length as the input vectors
- spmap_vec(): Returns an atomic vector with the same length as the input vectors
- spmap_lgl(): Returns a logical vector
- spmap_int(): Returns an integer vector
- spmap_dbl(): Returns a double vector
- spmap_chr(): Returns a character vector
- spmap_structure(): Returns a new glycan structure vector (.f must return igraph objects)

Value

- spmap(): A list
- spmap_vec(): An atomic vector of type specified by .ptype
- spmap_lgl/int/dbl/chr(): Atomic vectors of the corresponding type
- spmap_structure(): A new glyrepr_structure object

Examples

```
# Create structure vectors with duplicates
core1 <- o_glycan_core_1()
core2 <- n_glycan_core()
structures <- c(core1, core2, core1)  # core1 appears twice
weights <- c(1.0, 2.0, 1.0)  # corresponding weights
factors <- c(2, 3, 2)  # corresponding factors
```

```
# Map a function that uses structure, weight, and factor
spmap_dbl(list(structures, weights, factors),
          function(g, w, f) igraph::vcount(g) * w * f)

# Use purrr-style lambda functions
spmap_dbl(list(structures, weights, factors), ~ igraph::vcount(..1) * ..2 * ..3)

# Test with recycling
spmap_dbl(list(structures, 2.0, 3), ~ igraph::vcount(..1) * ..2 * ..3)
```

`structure_candidate_edges` *List Potential Virtual Edges for Floating Parts*

Description

`structure_candidate_edges()` represents every potential floating-part attachment as an explicit virtual edge. `from_node` is a candidate parent in another floating component or the main tree, and `to_node` is the root of the floating part.

The rows correspond one-to-one with the floating-part rows from `structure_floating_candidates()`. Floating substituents do not create virtual graph edges. For unrestricted `{<floating>}` parts, every feasible node outside the part's own component is returned and `scope` is `"all"`. For explicitly restricted parts, only the declared parent nodes are returned and `scope` is `"explicit"`.

Node indices refer to `structure_nodes()$node_id` for the same glycan. Missing structures and structures without floating parts contribute no rows. Duplicate structures are expanded to their original vector positions. For graph input, node indices are current numeric vertex positions and `glycan_id` is `1L`. If vector input is named, the result also contains a `glycan_name` column.

Usage

```
structure_candidate_edges(x)
```

Arguments

x	A glycan structure vector or one glycan `igraph`.

Value

A tibble with columns `glycan_id`, `part_id`, `from_node`, `to_node`, `linkage`, and `scope`, plus `glycan_name` when `x` is named.

Examples

```
glycan <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
structure_candidate_edges(glycan)
```

`structure_component_membership`
Identify Main and Floating Structure Components

Description

`structure_component_membership()` identifies whether each glycan node belongs to the main tree or to a floating part. It provides a public, normalized alternative to reading the private floating-part graph attribute.

Main-tree nodes have `component_type = "main"` and a missing `part_id`. Nodes in a floating subtree have `component_type = "floating"` and the corresponding `part_id` from `structure_floating_parts()`. Floating substituents do not introduce vertices, so they do not create additional component-membership rows.

Node indices refer to `structure_nodes()$node_id` for the same glycan. Missing structures contribute no rows. Duplicate structures are expanded to their original vector positions. For graph input, node indices are current numeric vertex positions and `glycan_id` is `1L`. If vector input is named, the result also contains a `glycan_name` column.

Usage

```
structure_component_membership(x)
```

Arguments

`x`	A glycan structure vector or one glycan `igraph`.

Value

A tibble with columns `glycan_id`, `node_id`, `component_type`, and `part_id`, plus `glycan_name` when `x` is named.

Examples

```
glycan <- as_glycan_structure(
  "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
structure_component_membership(glycan)
```

structure_floating_candidates
List Candidate Parents for Floating Metadata

Description

structure_floating_candidates() expands floating-part and floating-substituent metadata to one row per candidate parent. This provides a uniform representation for explicitly restricted and unrestricted floating metadata.

For an unrestricted floating part, every feasible node outside its own component is returned; for an unrestricted floating substituent, every feasible residue node in the complete structure is returned. These rows use scope = "all". For metadata written with an explicit |<parents> suffix, only the declared parent nodes are returned and scope is "explicit".

Floating-part rows have a non-missing part_id, root_node, and linkage. Floating-substituent rows instead have a non-missing substituent_id and substituent. Exactly one of part_id and substituent_id is non-missing in each row.

Node indices refer to structure_nodes()$node_id for the same glycan. Missing structures and structures without floating metadata contribute no rows. Duplicate structures are expanded to their original vector positions. For graph input, node indices are current numeric vertex positions and glycan_id is 1L. If vector input is named, the result also contains a glycan_name column.

Usage

```
structure_floating_candidates(x)
```

Arguments

x	A glycan structure vector or one glycan igraph.

Value

A tibble with columns glycan_id, part_id, root_node, parent_node, linkage, scope, substituent_id, and substituent, plus glycan_name when x is named.

Examples

```
glycans <- as_glycan_structure(c(
  unrestricted = "{Neu5Ac(a2-3)}Gal(b1-3)GalNAc(a1-",
  restricted = "{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-",
  substituent = "{6S}Gal(a1-3)Gal(a1-"
))
structure_floating_candidates(glycans)
```

`structure_from_arrays` *Construct Structures from Residue and Edge Arrays*

Description

A format-independent entry point for parsers. Node IDs are one-based positions in `mono`; edges point from parent to child. Input order need not be canonical. Each record contains `mono`, `sub`, `edges`, `linkage`, and `anomer`. `edges` is an interleaved integer vector `c(parent, child, ...)`. `sub` contains comma-separated substituent tokens (or empty strings). Optional `alditol` defaults to `FALSE`.

Usage

```
structure_from_arrays(
  x,
  on_failure = c("error", "na"),
  progress = FALSE,
  warn = TRUE
)
```

Arguments

`x`	A list of structure records. A `NULL` element represents a missing structure. Names, missing positions, and duplicate positions are preserved.
`on_failure`	Either `"error"` (default) or `"na"`. The latter warns and returns missing structures at invalid positions.
`progress`	`FALSE` (default) for silent operation, `TRUE` for a cli progress bar, or a function with arguments `stage`, `current`, and `total`. A callback owns its display; no cli bar is created. Counts are local to each stage, not an overall percentage. Character parsing counts unique non-missing inputs; graph construction may count deduplicated canonical structures. Updates are throttled, with stage boundaries always reported. Empty or already-converted inputs may report no stages. Callback errors abort the operation, including with `on_failure = "na"`. The cli bar is closed when the call exits, including on errors or interrupts, and follows cli display settings (short or non-interactive calls may show no visible bar).
`warn`	Whether to warn about invalid records when `on_failure = "na"`. Validation is always performed. Set to `FALSE` when the caller handles missing results itself.

Details

Optional `floating_parts` is a list of records containing `root`, `nodes`, `linkage`, and `parents`. Each `nodes` vector must contain exactly the nodes in that disconnected component. Optional `floating_substituents` is a list of records containing `substituent` and `parents`. Empty `parents` means all feasible nodes; all indices refer to the original input arrays. Singleton candidates are resolved during canonicalization.

Records above the native backend's size guard use the graph reference path; the guard is not an input-size limit. Only failing or unsupported records use that path. No format-specific strings are generated and reparsed. Failures have class `glyrepr_error_structure_failure` and fields `position`, `input_name`, and `reason`. Recovery warnings have class `glyrepr_warning_structure_failure` and fields `positions` and `reasons`.

Value

A `glyrepr_structure` vector, with graph storage deduplicated by canonical IUPAC key.

Examples

```
structure_from_arrays(list(list(
  mono = c("Glc", "Gal"), sub = c("", ""),
  edges = c(1L, 2L), linkage = "b1-4", anomer = "?1"
)))
```

---

structure_tables *Convert Glycan Structures to Graph Tables*

Description

`structure_nodes()`, `structure_edges()`, `structure_floating_parts()`, and `structure_floating_substituents()` convert a glycan structure vector or one glycan `igraph` to normalized graph tables. `structure_from_tibbles()` rebuilds a `glyrepr_structure` vector from those tables, a vector of reducing-end anomers, and optional alditol status.

The `glycan_id` column is the integer position of each glycan in the input vector. Duplicate structures are expanded to their original vector positions. Missing structures have no node or edge rows and are reconstructed from missing values in `anomers`. If `x` is named, all four tibbles also contain a `glycan_name` column. `structure_from_tibbles()` uses `glycan_name` as output names when that column is present.

For structure-vector input, `structure_nodes()$node_id` follows residue order in the complete canonical IUPAC-condensed string. For graph input, it follows the graph's current numeric vertex positions without canonicalizing or renumbering them. A graph is represented with `glycan_id = 1L` and no `glycan_name` column.

In `structure_floating_parts()`, `root_node` and every integer in the `nodes` and `parents` list-columns refer to `structure_nodes()$node_id` for the same glycan. `nodes` contains every node in the floating component. An empty `parents` vector means all feasible nodes outside that component are candidates. The `linkage` column describes the virtual attachment to an unresolved parent; this attachment is intentionally absent from `structure_edges()`. During reconstruction, a row with exactly one effective candidate parent is normalized to an ordinary edge and is therefore absent from the resulting `structure_floating_parts()` table.

Parent indices written after `|` in an IUPAC-condensed floating part are complete-sequence node IDs, identical to `structure_nodes()$node_id` for a canonical structure. Residues in floating blocks precede the main tree, and substituent blocks contribute no nodes. `structure_from_tibbles()` expects these same global node IDs and preserves cross-component domains.

In `structure_floating_substituents()`, each row describes one unresolved substituent. `substituent` is its canonical position-and-name token, and the `parents` list-column contains candidate global node IDs. An empty vector means all feasible residue nodes are candidates. A singleton candidate is normalized into `structure_nodes()$sub`, so it does not remain in the floating-substituent table.

Usage

```
structure_nodes(x)

structure_edges(x)

structure_floating_parts(x)

structure_floating_substituents(x)

structure_from_tibbles(
  nodes,
  edges,
  anomers,
  floating_parts = NULL,
  floating_substituents = NULL,
  alditols = FALSE
)
```

Arguments

`x`	A glycan structure vector or one glycan `igraph`.
`nodes`	A data frame with columns `glycan_id`, `node_id`, `mono`, and `sub`, and optionally `glycan_name`.
`edges`	A data frame with columns `glycan_id`, `edge_id`, `from_node`, `to_node`, and `linkage`, and optionally `glycan_name`.
`anomers`	A character vector of reducing-end anomers, one per glycan.
`floating_parts`	A data frame returned by `structure_floating_parts()`, or `NULL` when no floating parts are present.
`floating_substituents`	A data frame returned by `structure_floating_substituents()`, or `NULL` when no floating substituents are present.
`alditols`	A logical vector indicating alditol status, either one value or one per glycan. Missing values are allowed only for missing glycans. Defaults to `FALSE` for backward compatibility.

Value

- `structure_nodes()` returns a tibble with columns `glycan_id`, `node_id`, `mono`, and `sub`.
- `structure_edges()` returns a tibble with columns `glycan_id`, `edge_id`, `from_node`, `to_node`, and `linkage`.

- `structure_floating_parts()` returns a tibble with columns `glycan_id`, `part_id`, `root_node`, the list-column `nodes`, `linkage`, and the list-column `parents`.
- `structure_floating_substituents()` returns a tibble with columns `glycan_id`, `substituent_id`, `substituent`, and the list-column `parents`.
- `structure_from_tibbles()` returns a `glyrepr_structure` vector.

Examples

```
glycans <- c(o_glycan_core_1(), o_glycan_core_1())
nodes <- structure_nodes(glycans)
edges <- structure_edges(glycans)
structure_from_tibbles(nodes, edges, get_anomer(glycans))

floating <- as_glycan_structure(
  "{6S|1,2}{Neu5Ac(a2-6)|2,3}Gal(b1-3)GalNAc(a1-"
)
floating_parts <- structure_floating_parts(floating)
floating_substituents <- structure_floating_substituents(floating)
structure_from_tibbles(
  structure_nodes(floating),
  structure_edges(floating),
  get_anomer(floating),
  floating_parts,
  floating_substituents
)
```

---

structure_to_iupac *Convert Glycan Structure to IUPAC-like Sequence*

Description

Convert a glycan structure vector or one glycan `igraph` to a sequence representation in the form of mono(linkage)mono, with branches represented by square brackets []. The backbone is chosen as the longest path, and for branches, linkages are ordered lexicographically with smaller linkages on the backbone.

Usage

```
structure_to_iupac(glycan)
```

Arguments

`glycan`	A glyrepr_structure vector or one glycan `igraph`.

Value

A character vector for structure-vector input, or one unnamed character scalar for graph input.

Sequence Format

The sequence follows the format mono(linkage)mono, where:

- mono: monosaccharide name with optional substituents (e.g., Glc, GlcNAc, Glc3Me)
- linkage: glycosidic linkage (e.g., b1-4, a1-3)
- Branches are enclosed in square brackets []
- Substituents are appended directly to monosaccharide names (e.g., Glc3Me for Glc with 3Me substituent)
- An alditol reducing end has an `-ol` suffix before its reducing-end annotation (e.g., `GlcNAc-ol(a1-`)

Backbone Selection

The backbone is selected as the longest path in the tree. For branches, the same rule applies recursively.

Linkage Comparison

Linkages are compared lexicographically:

1. First by anomeric configuration: ? > b > a
2. Then by first position: ? > numbers (numerically)
3. Finally by second position: ? > numbers (numerically)

Smaller linkages are placed on the backbone, larger ones in branches.

For graph input, `structure_to_iupac()` validates and canonicalizes a copy of the graph before generating the sequence. Use `graph_to_iupac()` when the graph is already valid and canonical and the lower-level trusted-input path is desired.

Examples

```
# Simple linear structure
structure_to_iupac(o_glycan_core_1())

# Branched structure
structure_to_iupac(n_glycan_core())

# Structure with substituents
graph <- igraph::make_graph(~ 1-+2)
igraph::V(graph)$mono <- c("Glc", "GlcNAc")
igraph::V(graph)$sub <- c("3Me", "6Ac")
igraph::E(graph)$linkage <- "b1-4"
graph$anomer <- "a1"
glycan <- glycan_structure(graph)
structure_to_iupac(glycan)  # Returns "GlcNAc6Ac(b1-4)Glc3Me(a1-"
structure_to_iupac(graph)
structure_to_iupac(as_glycan_structure("GlcNAc-ol(a1-"))

# Vectorized structures
structs <- c(o_glycan_core_1(), n_glycan_core())
```

```
structure_to_iupac(structs)
```

`validate_glycan_graph` *Validate a Glycan Graph*

Description

Validate that a single `igraph` satisfies the structural and biochemical requirements of a glycan graph. The graph is returned unchanged.

Usage

```
validate_glycan_graph(graph)
```

Arguments

`graph`	A single `igraph` glycan graph.

Details

This function does not canonicalize the graph or generate an IUPAC-condensed string. Use `canonicalize_glycan_graph()` and `graph_to_iupac()` for those operations.

Value

`graph`, unchanged. An error is thrown when `graph` is invalid.

Choosing a construction pipeline

Prefer `as_glycan_structure()` for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use `structure_from_arrays()` to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, `canonicalize_glycan_graphs()` validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. `structure_from_tibbles()` converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. `validate_glycan_graph()` validates one graph; `canonicalize_glycan_graph()` assumes a valid graph; `graph_to_iupac()` assumes a valid, canonical graph. `validate_glycan_graph_vector()` checks the graph-list container, not each graph's semantics. `new_glycan_structure()` trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set `on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

Other low-level glycan structure functions: `canonicalize_glycan_graph()`, `graph_to_iupac()`, `new_glycan_structure()`, `validate_glycan_graph_vector()`

`validate_glycan_graph_vector` *Validate a List of Glycan Graphs*

Description

Check the container used to store individually valid glycan graphs in one glycan structure vector. Generic and concrete residues may coexist within a graph and across graphs.

Usage

```
validate_glycan_graph_vector(graphs, label = NULL)
```

Arguments

`graphs`	A list of individually valid `igraph` glycan graphs.
`label`	An optional label retained for backward compatibility.

Details

This function assumes that every element has already passed `validate_glycan_graph()`. It does not repeat scalar graph validation.

Value

`NULL`, invisibly.

Choosing a construction pipeline

Prefer `as_glycan_structure()` for ordinary construction from IUPAC strings or graphs. Parsers that already produce residue and edge arrays should use `structure_from_arrays()` to validate and canonicalize records through the shared compact backend, then build deduplicated graph storage.

For existing graph parsers, `canonicalize_glycan_graphs()` validates a batch and returns aligned canonical graphs, IUPAC keys, and per-element status. It preserves input names and arbitrary graph, vertex, and edge attributes. Equal structures retain separate graphs until the caller explicitly deduplicates them. This is useful when source attributes differ. Graph and array construction share native processing. Unsupported inputs use the R path, which also supplies the established failure diagnostics. `structure_from_tibbles()` converts table rows directly to array records.

The individual low-level functions remain available when an intermediate graph is needed. `validate_glycan_graph()` validates one graph; `canonicalize_glycan_graph()` assumes a valid graph; `graph_to_iupac()` assumes a valid, canonical graph. `validate_glycan_graph_vector()` checks the graph-list container, not each graph's semantics. `new_glycan_structure()` trusts that graphs are valid and canonical and match their keys; it does not validate, canonicalize, or deduplicate them. Incorrect use can create vectors that fail in later operations.

Floating graph schemas

A floating structure is one weakly disconnected graph with exactly one main outward tree and one outward tree per floating part. Its `floating_parts` graph attribute is a list of entries with integer `root`, integer `nodes`, character `linkage`, and integer `parents` fields. `nodes` contains every vertex in that floating component. `parents = integer()` means all feasible nodes outside that component, including nodes in other floating components. Legacy input graphs may omit `nodes`; canonical output graphs always contain it. During canonicalization, a part with exactly one effective candidate parent is converted to an ordinary graph edge and its floating metadata is removed. Otherwise, the virtual attachment linkage is not a graph edge. See `glycan_structure()` for the complete contract.

A graph may also have a `floating_substituents` attribute. It is a list of entries with character `substituent` and integer `parents` fields. An empty parent vector means all feasible residue nodes in the complete structure. A singleton candidate is moved into the selected vertex's `sub` attribute during canonicalization. All parent indices refer to the complete graph's vertex positions, not positions within a component.

Name-preserving graph construction

Use the batch interface to obtain canonical graphs and keys together, then explicitly deduplicate graph storage when constructing a structure vector:

```
batch <- canonicalize_glycan_graphs(graphs)
keep <- batch$status == "ok" & !duplicated(batch$iupac)
unique_graphs <- batch$graphs[keep]
names(unique_graphs) <- unname(batch$iupac[keep])

new_glycan_structure(batch$iupac, unique_graphs)
```

Names and missing positions are preserved; use `NULL` for missing input graphs. With the default `on_failure = "error"`, an invalid graph raises an error identifying its original position. Set `on_failure = "na"` to warn and retain successful entries, with invalid positions represented by missing keys and `NULL` graphs. The same assembly code works in both modes. Deduplication keeps the first graph for each key, including its source attributes. Keep `batch$graphs` instead when per-input provenance is needed. Use `validate = FALSE` only when every non-missing graph has already passed `validate_glycan_graph()`.

Both `as_glycan_structure()` and `structure_from_arrays()` accept `progress = TRUE` for stage-level progress. A parser that owns its progress display can instead pass a function `function(stage, current, total)` to receive updates without creating a nested bar. Counts restart for each stage; completion of parsing alone does not mean structure construction is finished.

See Also

Other low-level glycan structure functions: `canonicalize_glycan_graph()`, `graph_to_iupac()`, `new_glycan_structure()`, `validate_glycan_graph()`

`valid_linkages` *Check if Linkages are Valid*

Description

Valid linkages are in the form of "a1-2", "b1-4", "a?-1", etc. Specifically, the pattern is `xy-z`:

- `x`: the anomer, either "a", "b", or "?".
- `y`: the first position, either "1", "2" or "?".
- `z`: the second position, either a 1-9 digit or "?". Can also be multiple positions separated by "/", e.g. "1/2/3". "?" could not be used with "/".

Usage

```
valid_linkages(linkages)
```

Arguments

`linkages`	A character vector of linkages.

Value

A logical vector.

Examples

```
# Valid linkages
valid_linkages(c("a1-2", "?1-4", "a?-1", "b?-?", "??-?", "a1/2-3"))

# Invalid linkages
valid_linkages(c("a1-2/?", "1-4", "a/b1-2", "c1-2", "a9-1"))
```

Index

∗ **low-level glycan structure functions**
canonicalize_glycan_graph, 7
graph_to_iupac, 27
new_glycan_structure, 35
validate_glycan_graph, 58
validate_glycan_graph_vector, 60

as_glycan_composition, 3
as_glycan_structure, 4
as_glycan_structure(), *8, 9*, *14, 15*, *27, 28*, *30, 31*, *35, 36*, *58–61*
available_monosaccharides, 6
available_monosaccharides(), *11*, *20*
available_substituents, 7

canonicalize_glycan_graph, 7, *28*, *36*, *59*, *61*
canonicalize_glycan_graph(), *8*, *14*, *27*, *35*, *58*, *60*
canonicalize_glycan_graphs, 9
canonicalize_glycan_graphs(), *8*, *14*, *27*, *35*, *58*, *60*
convert_to_generic, 10
convert_to_generic(), *20*
count_mono, 12

enumerate_floating_graph_localizations, 13
enumerate_floating_localizations, 15
enumerate_floating_localizations(), *13*, *15*

fill_anomer_pos, 17

get_alditol, 18
get_anomer, 18
get_anomer_pos *(*infer_anomer_pos*)*, 32
get_mono_type, 19
get_mono_type(), *6*, *22*
get_structure_graphs, 20
get_structure_level, 21
glycan_composition, 22
glycan_composition(), *3*
glycan_structure, 23
glycan_structure(), *8*, *14*, *17*, *20, 21*, *25*, *28–30*, *32*, *36, 37*, *59*, *61*
graph_to_iupac, *9*, 27, *36*, *59*, *61*
graph_to_iupac(), *8*, *14*, *27*, *35*, *57, 58*, *60*

has_floating_parts, 29
has_floating_parts(), *22*, *31*
has_floating_substituents, 30
has_floating_substituents(), *22*, *30*
has_linkages, 31
has_linkages(), *22*, *39*

infer_anomer_pos, 32
is_glycan_composition *(*glycan_composition*)*, 22
is_glycan_structure *(*glycan_structure*)*, 23
is_known_monosaccharide, 33

localize_floating_parts, 33

n_glycan_core, 37
new_glycan_structure, *9*, *28*, 35, *59*, *61*
new_glycan_structure(), *8*, *14*, *27*, *35*, *58*, *60*

o_glycan_core_1 *(*n_glycan_core*)*, 37
o_glycan_core_2 *(*n_glycan_core*)*, 37

possible_linkages, 38
possible_linkages(), *32*

remove_linkages, 39
remove_linkages(), *32*, *39*
remove_substituents, 40

severy *(*smap_predicates*)*, 46
simap, 40

simap_chr (simap), 40
simap_dbl (simap), 40
simap_int (simap), 40
simap_lgl (simap), 40
simap_structure (simap), 40
simap_vec (simap), 40
smap, 42
smap2, 44
smap2_chr (smap2), 44
smap2_dbl (smap2), 44
smap2_int (smap2), 44
smap2_lgl (smap2), 44
smap2_structure (smap2), 44
smap2_vec (smap2), 44
smap_chr (smap), 42
smap_dbl (smap), 42
smap_int (smap), 42
smap_lgl (smap), 42
smap_predicates, 46
smap_structure (smap), 42
smap_unique, 47
smap_vec (smap), 42
snone (smap_predicates), 46
spmap, 48
spmap_chr (spmap), 48
spmap_dbl (spmap), 48
spmap_int (spmap), 48
spmap_lgl (spmap), 48
spmap_structure (spmap), 48
spmap_vec (spmap), 48
ssome (smap_predicates), 46
structure_candidate_edges, 50
structure_component_membership, 51
structure_component_membership(), *29*
structure_edges (structure_tables), 54
structure_floating_candidates, 52
structure_floating_candidates(), *34*, *50*
structure_floating_parts (structure_tables), 54
structure_floating_parts(), *29, 30*, *51*
structure_floating_substituents (structure_tables), 54
structure_floating_substituents(), *31*
structure_from_arrays, 53
structure_from_arrays(), *8, 9*, *14, 15*, *27, 28*, *35, 36*, *58–61*
structure_from_tibbles (structure_tables), 54
structure_from_tibbles(), *8*, *14*, *27*, *35*, *58*, *60*
structure_nodes (structure_tables), 54
structure_tables, 54
structure_to_iupac, 56

valid_linkages, 62
valid_linkages(), *38, 39*
validate_glycan_graph, *9*, *28*, *36*, 58, *61*
validate_glycan_graph(), *7–9*, *14, 15*, *27, 28*, *35, 36*, *58–61*
validate_glycan_graph_vector, *9*, *28*, *36*, *59*, 60
validate_glycan_graph_vector(), *8*, *14*, *27*, *35*, *58*, *60*