Roan

Version 8.0.7

Table of Contents

1 Introduction

Roan is a library of Common Lisp (`http://en.wikipedia.org/wiki/Common_Lisp`) code for writing applications related to change ringing (`http://www.ringing.org/change-ringing`). It is roughly comparable to the Ringing Class Library (`http://ringing-lib.sourceforge.net/`), although that is for the C++ programming language, and the two libraries differ in many other ways.

Roan provides

- facilities for representing rows and changes as Lisp objects, and permuting them, etc.
- functions for reading and writing place notation, extended to support jump changes as well as conveniently representing palindromic sequences of changes
- a set data structure suitable for collecting and proving sets of rows, or sets of sets of rows
- a pattern language for matching rows, for example, for identifying ones with properties considered musically desirable; and that includes the ability to match pairs of rows, which enables identifying wraps
- a data structure for describing methods, which can include jump changes,
- a searchable database of method definitions, together with a mechanism for updating that database from the ringing.org web site (`http://www.ringing.org/`)
- a function for extracting false course heads from common kinds of methods
- there's more in the works, just not ready for public release yet See [future-plans], page 54.

While this manual describes Roan, it is neither a tutorial on Lisp nor one on change ringing. If you don't know Common Lisp or don't know about change ringing, much of this manual is likely to be confusing.

Roan is distributed under an MIT open source license. While you should read it for complete details, it largely means that you can just use Roan for nearly anything you like. See Appendix A [License], page 52. Roan also loads and uses a variety of other libraries. See [dependencies], page 53.

1.1 Obtaining and installing Roan

While Quicklisp (`http://quicklisp.org`) is not required to run Roan, it is recommended. With Quicklisp installed and configured, you can download and install Roan by simply evaluating `(ql:quickload :roan)`.

Quicklisp's `quickload` function, above, will also pull in all the other libraries upon which Roan depends; if you don't use Quicklisp you will have to ensure that those libraries are available and loaded. If you don't want to use Quicklisp, and prefer to load Roan by hand, the repository for Roan itself is at `https://bitbucket.org/dfmorrison/roan`, and both current and previous versions can be downloaded from the tags pane of the Downloads page, `https://bitbucket.org/dfmorrison/roan/downloads/?tab=tags`.

One of the libraries on which Roan depends, `cl-sqlite`, depends upon a C language library, `libsqlite3`, already being installed. It may already be on your system, but if not it is readily available from the SQLite web site (`https://www.sqlite.org/`). When

loading Roan, below, if this library is not present an error will be signaled, and you will typically be thrown into your Lisp's debugger when Quicklisp is loading `cl-sqlite`. If for some reason you prefer not to install `libsqlite` there is a version of Roan with slightly reduced functionality, `roan-base`, that does not contain the method lookup functions and does not load `cl-sqlite`. To use it instead simply evaluate `(ql:quickload :roan-base)`. See Section 5.2 [Methods database], page 44.

Roan has been tested with

- CCL (Clozure CL) (`http:ccl.clozure.com`) version 1.11 (64 bit), on Ubuntu Linux 16.04 and macOS 10.13.5
- SBCL (Steel Bank Common Lisp) (`http://sbcl.org`) (64 bit) version 1.3.18 on Ubuntu Linux 16.04 and version 1.2.11 on macOS 10.13.5
- CLISP (`http://clisp.org`), version 2.49 on Ubuntu Linux 16.04

but should also work in other, modern Common Lisp implementations that support the libraries on which Roan depends. See [dependencies], page 53.

One function, `update-methods-database`, does not work in CLISP or LispWorks (`http://www.lispworks.com`) because the libraries used to download and unzip the updated database currently (as of June 2017) do not work or cannot be loaded in those Lisp implementations. See [update-methods-database], page 48, for further details and workarounds.

1.2 Reporting bugs

The best way to report bugs is to submit them with Roan's Bitbucket issue tracker (`https://bitbucket.org/dfmorrison/roan/issues`). If that doesn't work for you you can also send mail to Don Morrison `<dfm@ringing.org>`.

It would be helpful, and will be more likely to lead to successful resolution of the bug, if a bug report includes

- a detailed prescription of how generate the bug, preferably from as simple a starting place as you can use to reproduce it; that is, send code, that when evaluated, demonstrates the bug
- the version of Roan you are using; this can be found by evaluating `(asdf:component-version (asdf:find-system :roan))`
- the name and version number of the Lisp implementation you are using, as well as whether it is 32-bit or 64-bit if both are available
- the name and version number of the operating system on which it is running
- the kind of processor on which it is running, especially if it's something unusual

1.3 A note on examples

Examples in this manual are typically of the form

```
(caddr '(1 2 3 4)) ⇒  3
```

That is, an expression, followed by '⇒' and a printed representation of the result of evaluating that expression. That right hand side is typically not exactly as the REPL (Read Eval Print Loop) might print it: for example, symbols will usually be shown in lower case

while most Lisp implementation's REPLs will use upper case; and things like hash-sets that have indeterminate order may result in different orders of elements of lists.

Occasionally, though, examples will look like

```
CL-USER> (+ 1 2 3)
6
CL-USER> (values (+ 1 2 3) (cons 'a 'b))
6
(A . B)
CL-USER>
```

In this case the example is a transcript of an interaction with a REPL. None of the examples makes explicit note of which of these two styles is being used, it being assumed the reader can easily deduce this from their appearances.

1.4 Multi-threading

If the symbol `:bordeaux-threads` is in `*features*` *before* Roan is loaded, Roan will try to ensure that all its calls can be used safely in multiple threads simultaneously, using the `bordeaux-threads` API. This has not yet been extensively tested, however.

It is only safe to use multiple threads with `bordeaux-threads`; using Roan in multiple threads in an implementation's native threading API, without loading `bordeaux-threads` before Roan, will likely fail.

1.5 The roan package

All the symbols used by Roan to name functions, variables and so on are in the `roan` package. When using them from another package, such as `cl-user`, they should be prefixed with an explicit `roan:`.

```
CL-USER> *package*
#<Package "COMMON-LISP-USER">
CL-USER> roan:+maximum-stage+
24
```

Alternatively all the external symbols of the `roan` package can be imported into a package with `use-package`, or the `:use` option to `defpackage`. There is the slight complication, however, that the `roan` package shadows the symbols `method` and `method-name` from the `common-lisp` package. This is done because methods are an important concept in change ringing, albeit one unrelated to CLOS methods. Typically `method` and `method-name` should be shadowed in other packages that use the `roan` package. This can be done with `shadowing-import`, or the `:shadowing-import` option to `defpackage`.

```
MY-PACKAGE> *package*
#<Package "MY-PACKAGE">
MY-PACKAGE> (package-use-list *)
(#<Package "COMMON-LISP">)
MY-PACKAGE> (shadowing-import '(roan:method roan:method-name))
T
MY-PACKAGE> (use-package :roan)
T
MY-PACKAGE> +maximum-stage+
24
```

`roan` [Package]

Contains the symbols used by Roan. The `roan` package shadows two symbols from the `common-lisp` package: `method` and `method-name`. The functions and so on attached to these symbols in the `common-lisp` package are usually only needed when doing introspection, and the shadowing should rarely cause difficulties.

`use-roan-package` **&optional** *package* [Function]

A convenience function for using the `roan` package. Causes *package*, which defaults to the current value of `*package*`, to inherit all the external symbols of the `roan` package, and shadows `method` and `method-name`. Signals a `type-error` if *package* is not a package designator. Signals a `package-error` if *package* is the `keyword` package.

```
MY-PACKAGE> *package*
#<Package "MY-PACKAGE">
MY-PACKAGE> (package-use-list *)
(#<Package "COMMON-LISP">)
MY-PACKAGE> (roan:use-roan-package)
T
MY-PACKAGE> +maximum-stage+
24
```

2 Fundamental Types

Central to change ringing is permuting sequences of a fixed collection of bells, where the cardinality of that collection is the stage. For modeling such things Roan provides the types `bell`, `stage` and `row`, and various operations on them. It is also provides tools for reading and writing place notation.

2.1 Bells

Roan supports ringing on as few as 2, or as many as 24, bells. Bells are represented as small, non-negative integers less than this maximum stage. However, bells as the integers used in Roan are zero-based: the treble is zero, the tenor on eight is 7, and so on. The `bell` type corresponds to integers in this range. There are functions for mapping bells to and from the characters corresponding to their usual textual representation in change ringing.

`bell` [Type]

A representation of a bell. These are zero-based, small integers, so the treble is `0`, the second is `1`, up to the tenor is one less than the stage.

`bell-name` *bell* **&optional** *upper-case* [Function]

Returns a character denoting this *bell*, or `nil` if *bell* is not a `bell`. If the character is alphabetic, an upper case letter is return if the generalized boolean *upper-case* is true, and otherwise a lower case letter. If *upper-case* is not supplied it defaults to the current value of `*print-bells-upper-case*`.

```
(bell-name 0) ⇒ #\1
(map 'string #'bell-name
     (loop for i from 0 below +maximum-stage+
           collect i))
    ⇒ "1234567890ETABCDFGHJKLMN"
(bell-name -1) ⇒ nil
(bell-name +maximum-stage+) ⇒ nil
```

`bell-from-name` *char* [Function]

Returns the `bell` denoted by the character designator *char*, or `nil` if it is not a character designator denoting a bell. The determination is case-insensitive.

```
(bell-from-name "8") ⇒ 7
(bell-from-name "E") ⇒ 10
(map 'list #'bell-from-name "135246") ⇒ (0 2 4 1 3 5)
(bell-from-name "%") ⇒ nil
```

`*print-bells-upper-case*` [Variable]

When printing bell names that are letters, whether or not to use upper case letters by default. It is a generalized boolean, with an initial default value of `t`.

2.2 Stages

The `stage` type represents the subset of small, positive integers corresponding to the numbers of bells Roan supports. While Roan represents stages as small, positive integers, it is

conventional in ringing to refer to them by names, such as “Minor” or “Caters”. There are functions for mapping stages, the integers used by Roan, to and from their conventional names as strings.

`stage` [Type]
A supported number of bells, an integer between `+minimum-stage+` and `+maximum-stage+`, inclusive.

`+minimum-stage+` [Constant]
The smallest number of bells supported, 2.

`+maximum-stage+` [Constant]
The largest number of bells supported, 24.

`stage-name` *stage* [Function]
Returns a string, the conventional name for this *stage*, capitalized, or `nil` if *stage* is not an integer corresponding to a supported stage.

```
(stage-name 8) ⇒ "Major"
(stage-name 22) ⇒ "Twenty-two"
(stage-name (1+ +maximum-stage+)) ⇒ nil
```

`stage-from-name` *name* [Function]
Returns a stage, a small, positive integer, with its name the same as the string designator *name*, or, if there is no stage with such a name, `nil`. The determination is made case-insensitively.

```
(stage-from-name "cinques") ⇒ 11
(stage-from-name "no-such-stage") ⇒ nil
```

`*default-stage*` [Variable]
An integer, the default value for optional or keyword arguments to many functions that must have a stage specified. See [`write-row`], page 7, [`row-string`], page 8, [`write-place-notation`], page 16, and [`place-notation-string`], page 17.

2.3 Rows

The fundamental units of ringing are rows and changes, permutations of a fixed set of bells. A distinction between them is often made, where a row is a permutation of bells and a change is a permutation taking one row to the next. In Roan they are both represented by the same data type, `row`; `row`s should be treated as immutable.

The Lisp reader is augmented by Roan to read `row`s printed in the notation usually used by change ringers by using the ‘`!`’ reader macro. For example, queens on twelve can be entered in Lisp as `!13579E24680T`. When read with the ‘`!`’ reader macro bells represented by alphabetic characters can be either upper or lower case; so queens on twelve can also be entered as `!13579e24680t` or `!13579e24680T`.

To support the common case of writing lead heads of treble dominated methods if the treble is leading it can be omitted. Thus, queens on twelve can also be entered as `!3579E24680T`. Apart from a leading treble, however, if any bell is omitted from a row written with a leading ‘`!`’ character an error will be signaled.

Note that `row`s are Lisp atoms, and thus literal values can be written using '`!`' notation without quoting, though quoting `row`s read that way will do no harm when they are evaluated.

Similarly, `row`s are printed using this same notation, `*print-escape*` controlling whether or not they are preceded by '`!`' characters. Note that the characters used to represent bells in this printed representation differ from the small integer `bell`s used to represent them internally, since the latter are zero based. For example, the treble is represented internally by the integer 0, but in this printed representation by the digit character '`1`'. When printing rows in this way a leading treble is not elided. And `*print-bells-upper-case*` can be used to control the case of bells in the printed representation of `row`s that are representated by letters, in cinques and above.

```
CL-USER> !12753468
!12753468
CL-USER> '!2753468
!12753468
CL-USER> (format t "with:    ~S~%without:  ~:*~A~%" !TE0987654123)
with:    !TE0987654123
without:  TE0987654123
NIL
CL-USER> (let ((roan:*print-bells-upper-case* nil))
           (format nil "~A" !TE0987654123))
"te0987654123"
CL-USER>
```

Rows can be compared for equality using `equalp` (but not `equal`). That is, two different `row` objects that correspond to the same ordering of the same number of bells will be `equalp`. Hash tables with a `:test` of `equalp` are often useful with `row`s. See [hash-set], page 19.

```
(equalp !13572468 !13572468) ⇒ t
(equalp !13572468 !12753468) ⇒ nil
(equalp !13572468 !1357246) ⇒ nil
(equalp !13572468 !3572468) ⇒ t
```

`row` [Type]

A permutation of bells at a particular stage. The type `row` is used to represent both change ringing rows and changes; that is, rows may be permuted by other rows. Instances of `row` should normally be treated as immutable.

`row-p` *object* [Function]

Non-nil if and only if *object* is a `row`.

`stage` *row* [Function]

The number of `bell`s of which the `row` *row* is a permutation.

`write-row` *row* **&key** *stream escape upper-case* [Function]

Writes *row*, which should be a `row`, to the indicated *stream*. The case of any bells represented by letters is controlled by *upper-case*, a generalized boolean defaulting to the current value of `*print-bells-upper-case*`. *escape*, a generalized Boolean defaulting to the current value of `*print-escape*`, determines whether or not to

write it in a form that read can understand. Signals a `type-error` if *row* is not a `row`, and the usual errors if *stream* is not open for writing, etc.

`row-string` *row* **&optional** *upper-case* [Function]
Returns a string representing the `row` *row*. The case of any bells represented by letters is controlled by *upper-case*, a generalized boolean defaulting to the current value of `*print-bells-upper-case*`. Signals a `type-error` if *row* is not a `row`.

`read-row` **&optional** *stream eof-error-p eof-value recursive-p* [Function]
Constructs and returns a `row` from the conventional symbols for bells read from the *stream*. The stage of the row read is determined by the bells present, that is by the largest bell for which a symbol is read. The treble can be elided, in which case it is assumed to be leading; a `parse-error` is signaled if any other bell is omitted. Bells represented by letters can be either upper or lower case.

`parse-row` *string* **&key** *start end junk-allowed* [Function]
Contructs a `row` from the conventional symbols for bells in the section of string *string* delimited by *start* and *end*, possibly preceded or followed by whitespace. The treble can be elided, in which case it is assumed to be leading; a `parse-error` is signaled if any other bell is omitted. Bells represented by letters can be either upper or lower case. If *string* is not a string a `type-error` is signaled. If the generalized boolean *junk-allowed* is false, the default, an error will be signaled if additional non-whitespace characters follow the representation of a row. Returns two values: the `row` read and a non-negative integer, the index into the string of the next character following all those that were parsed, including any trailing whitespace; if parsing consumed the whole of *string*, the second value will be length of *string*.

`row` **&rest** *bells* [Function]
Constructs and returns a `row` containing the *bells*, in the order they appear in the argument list. If the treble is not present, it defaults to being the first bell in the row. Duplicate bells or bells other than the treble missing result in an error being signaled.

```
(row 2 1 3 4 7 6 5) ⇒ !13245876
```

`rounds` **&optional** *stage* [Function]
Returns a row representing rounds at the given *stage*, which defaults to `*default-stage*` Signals a `type-error` if *stage* is not a `stage`, that is an integer between `+minimum-stage+` and `+maximum-stage+`, inclusive.

`bell-at-position` *row position* [Function]
`position-of-bell` *bell row* [Function]
The `bell-at-position` function returns the `bell` (that is, a small integer) at the given *position* in the *row*. The `position-of-bell` function returns position of *bell* in *row*, or `nil` if *bell* does not appear in *row*. The indexing into *row* is zero-based; so, for example, the leading bell is at position 0, not 1. Signals an error if *row* is not a `row`, or if *position* is not a non-negative integer or is too large for the stage of *row*

```
(bell-at-position !13572468 3) ⇒ 6
(bell-name (bell-at-position !13572468 3)
    ⇒ #\7
(position-of-bell 6 !13572468) ⇒ 3
(position-of-bell (bell-from-name #7) !13572468)
    ⇒ 3
```

`bells-list` *row* [Function]
`bells-vector` *row* **&optional** *vector* [Function]

The `bells-list` function returns a fresh list of `bell`s (small, non-negative integers, zero-based), the bells of *row*, in the same order that they appear in *row*. The `bells-vector` function returns a vector of `bell`s (small, non-negative integers, zero-based), the bells of *row*, in the same order that they appear in *row*. If *vector* is not supplied or is `nil` a freshly created, simple general vector is returned.

```
(bells-list !13572468) ⇒ (0 2 4 6 1 3 5 7)
(bells-vector !142536) ⇒ #(0 3 1 4 2 5)
```

If a non-nil *vector* is supplied the `bell`s are copied into it and it is returned. If *vector* is longer than the stage of *row* only the first elements of *vector*, as many as the stage of *row*, are over-written; the rest are unchanged. If *vector* is shorter than the stage of *row*, then, if it is adjustable, it is adjusted to be exactly as long as the stage of *row*, and otherwise an error is signaled without any modifications made to the contents of *vector* or its fill-pointer, if any. If *vector* has a fill-pointer and is long enough to hold all the bells of *row*, possibly after adjustment, its fill-pointer is set to the stage of *row*.

A `type-error` is signaled if *row* is not a `row`. An error is signled if *vector* is neither `nil` nor a `vector` with an element type that is a supertype of `bell`, and of sufficient length or adjustable.

2.3.1 Properties of rows

`roundsp` *row* [Function]

True if and only if *row* is a `row` representing rounds at its stage.

```
(roundsp !23456) ⇒ t
(roundsp !123546) ⇒ nil
(roundsp 123456) ⇒ nil
```

`changep` *row* [Function]

True if and only if *row* is a `row` representing a permutation with no bell moving more than one place.

```
(changep !214365) ⇒ t
(changep !143265) ⇒ nil
(changep |214365|) ⇒ nil
```

`placesp` *row* **&rest** *places* [Function]

Returns true if and only if *row* is a (non-jump) change, with exactly the specified *places* being made, and no others. To match a cross at even stages supply no *places*.

Signals a `type-error` if *row* is not a `row` or any of *places* are not `bell`s. Signals an `error` if any of *places* are not less than the stage of *row*, or are duplicated.

```
(placesp !21354768 2 7) ⇒ t
(placesp !21346587 2 7) ⇒ nil
(placesp !21354768 2) ⇒ nil
(placesp !2135476 2) ⇒ t
(placesp !21436587) ⇒ t
```

`in-course-p` *row* [Function]

True if and only if *row* is a `row` representing an even permutation.

```
(in-course-p !132546) ⇒ t
(in-course-p !214365) ⇒ nil
(in-course-p "132546") ⇒ nil
```

`involutionp` *row* [Function]

True if and only if *row* is a `row` that is its own inverse.

```
(involutionp !13248765) ⇒ t
(involutionp !13425678) ⇒ nil
(involutionp nil) ⇒ nil
```

`order` *row* [Function]

Returns a positive integer, the order of *row*: the minimum number of times it must be permuted by itself to produce rounds. A `type-error` is signaled if *row* is not a `row`.

```
(order !13527486) ⇒ 7
(order !31256784) ⇒ 15
(order !12345678) ⇒ 1
```

`cycles` *row* [Function]

Returns a list of lists of bells. Each of the sublists is the orbit of all of its elements in *row*. One cycles are included. Thus, if *row* is a lead head, all the sublists of length one are hunt bells, all the rest being working bells. If there are two or more sublists of length greater than one the corresponding method is, in Central Council nomenclature, a differential or differential hunter, depending upon the absence or presence of hunt bells. The resulting sublists are each ordered such that the first bell is the lowest numbered bell in that cycle, and the remaining bells occur in the order in which a bell traverses the cycle. Within the top level list, the sublists are ordered such that the first bell of each sublist appear in ascending numerical order.

```
(cycles !13572468) ⇒ ((0) (1 4 2) (3 5 6) (7))
(format nil "~{(~{~C~^,~})~^, ~}"
            (mapcar #'(lambda (x) (mapcar #'bell-name x))
            (cycles !13572468)))
    ⇒ "(1), (2,5,3), (4,6,7), (8)"
```

`tenors-fixed-p` *row* **&optional** *starting-at* [Function]

Returns true if and only if all the bells of *row* at positions *starting-at* or higher are in their rounds positions. In the degenerate case of *starting-at* being equal to or

greater than the stage of *row* it returns true. Note that it is equivalent to `(not (null (alter-stage row starting-at)))`. If not supplied *starting-at* defaults to `6`, that is the position of the bell conventionally called the seven, though represented in Roan by the small integer `6`. Signals a `type-error` if *row* is not a `row` or *starting-at* is not a non-negative integer.

```
(tenors-fixed-p !13254678) ⇒ t
(tenors-fixed-p !13254678 5) ⇒ t
(tenors-fixed-p !13254678 4) ⇒ nil
(tenors-fixed-p !54321) ⇒ t
(tenors-fixed-p !54321 4) ⇒ nil
```

`which-plain-bob-lead-head` *row* [Function]

If *row* is a lead head of a plain course of Plain Bob at its stage returns a positive integer identifying which lead head it is; returns `nil` if *row* is not a Plain Bob lead head. If *row* is the first lead head of a plain course of Plain Bob `1` is returned, if the second `2`, etc. For the purposes of this function rounds is not a Plain Bob lead head, nor is any row below minimus. Signals a `type-error` if *row* is not a `row`.

```
(which-plain-bob-lead-head !13527486) ⇒ 1
(which-plain-bob-lead-head !42638507T9E) ⇒ 10
(which-plain-bob-lead-head !129785634) ⇒ nil
(which-plain-bob-lead-head !12345) ⇒ nil
(which-plain-bob-lead-head !132) ⇒ nil
```

`which-grandsire-lead-head` *row* [Function]

If *row* is a lead head of a plain course of Grandsire at its stage returns a positive integer identifying which lead head it is; returns `nil` if *row* is not a Grandsire lead head. If *row* is the first lead head of a plain course of Grandsire `1` is returned, if the second `2`, etc. For the purposes of this function rounds is not a Grandsire lead head, nor is any row below doubles. Signals a `type-error` if *row* is not a `row`.

```
(which-plain-bob-lead-head !1253746) ⇒ 1
(which-plain-bob-lead-head !28967453) ⇒ 4
(which-plain-bob-lead-head !135264) ⇒ nil
(which-plain-bob-lead-head !12345) ⇒ nil
(which-plain-bob-lead-head !1243) ⇒ nil
```

`plain-bob-lead-end-p` *row* [Function]

Returns true if *row* is a lead end (that is, the handstroke of the treble's full lead) of a lead of a plain course of Plain Bob at its stage, and otherwise `nil`. For the purposes of this function no `row` below minimus can be a Plain Bob lead end. Signals a `type-error` if *row* is not a `row`.

```
(plain-bob-lead-end-p !124365) ⇒ t
(plain-bob-lead-end-p !674523) ⇒ t
(plain-bob-lead-end-p !13527486) ⇒ nil
```

2.3.2 Permuting rows

`permute` *row* **&rest** *changes* [Function]

Permutes *row* by the *changes* in turn. That is, *row* is first permuted by the first of the *changes*, then the resuling row is permuted by second of the *changes*, and so on. Returns the row resulting from applying all the changes. So long as one or more *changes* are supplied the returned `row` is always a freshly created one: *row* and none of the *changes* are modified (as you'd expect, since they are intended to be viewed as immutable). The *row* and all the *changes* should be `row`s.

At each step of permuting a row by a change, if the row is of higher stage than the change, only the first *stage* bells of the row are permuted, where *stage* is the stage of the change, all the remaining bells of the row being unmoved. If the row is of lower stage than the change, it is as if the row were extended with bells in their rounds' positions for all the bells *stage* and above. Thus the result of each permuation step is a `row` whose stage is the larger of those of the row and the change.

If no *changes* are supplied `row` is returned. Signals a `type-error` if *row* or any of the *changes* are not `row`s.

```
(permute !34256 !35264) ⇒ !145362
(permute !34125 !4321 !1342) ⇒ !24315
(permute !4321 !654321) ⇒ !651234
(let ((r !13572468))
  (list (eq (permute r) r)
        (equalp (permute r (rounds 8)) r)
        (eq (permute r (rounds 8)) r)))
    ⇒ (t t nil)
```

`permute-collection` *collection change* [Function]
`permute-by-collection` *row collection* [Function]
`npermute-collection` *collection change* [Function]
`npermute-by-collection` *row collection* [Function]

Permutes each of the elements of a sequence or `hash-set` and an individual `row`, collecting the results into a similar collection. The `permute-collection` version permutes each the elements of *collection* by *change*; `permute-by-collection` permutes *row* by each of the elements of *collection* by *change*. The return value is a list, vector or `hash-set` if *collection* is a list, vector or `hash-set`, respectively. The `permute-collection` and `permute-by-collection` versions always return a fresh collection; the `npermute-collection` and `npermute-by-collection` versions modify *collection*, replacing its contents by the permuted rows. If *collection* is a sequence the contents of the result are in the same order: that is, the Nth element of the result is the Nth element supplied in *collection* permuted by or permuting *change* or *row*. If *collection* is a vector, `permute-collection` and `permute-by-collection` always return a simple, general vector.

If the result is a sequence, or if all the elements of *collection* were of the same stage as one another, it is guaranteed that the result will be the same length or cardinality as *collection*. However, if *collection* is a `hash-set` containing rows of different stages the result may be of lower cardinality than then the supplied `hash-set`, if *collection* contained two or more elements that were not `equalp` because they were of different

stages, but after being permuted by, or permuting, a higher stage row the results are `equalp`.

Signals a `type-error` if *change*, *row* or any of the elements of *collection* are not `rows`s, or if *collection* is not a sequence or `hash-set`.

`generate-rows` *changes* **&optional** *initial-row* [Function]
`ngenerate-rows` *changes* **&optional** *initial-row* [Function]

Generates a sequence of `rows` by permuting a starting `row` successively by each element of the sequence *changes*. The elements of *changes* should be `rows`. If *initial-row* is supplied it should be a `row`. If it is not supplied, rounds at the same stage as the first element of *changes* is used; if *changes* is empty, rounds at `*default-stage*` is used. Two values are returned. The first is a sequence of the same length as *changes*, and the second is a `row`. So long as *changes* is not empty, the first element of the first return value is *initial-row*, or the default rounds. The next value is that `row` permuted by the first element of *changes*; then that `row` permuted by the next element of *changes*, and so on, until all but the last element of *changes* has been used. The second return value is the last element of the first return value permuted by the last element of *changes*. If *changes* is empty, then the first return value is also empty, and *initial-row*, or the default rounds, is the second return value. Thus, for most methods, if *changes* are the changes of a lead, the first return value will be the rows of a lead starting with *initial-row*, and the second return value the lead head of the following lead.

If *changes* is a list, the first return value is a list; if *changes* is a vector, the first return value is a vector. The `generate-rows` function always returns a fresh sequence as its first return value, while `ngenerate-rows` resuses *changes*, replacing its elements by the permuted rows and returning it. The fresh vector created and returned by `generate-rows` is always a simple, general vector.

Signals an error if *initial-row* is neither a `row` nor `nil`, if *changes* isn't a sequence, or if any elements of *changes* are not `rows`.

```
(multiple-value-list
  (generate-rows '(!2143 !1324 !2143 !1324) !4321))
    ⇒ ((!4321 !3412 !3142 !1324) !1234)
```

`inverse` *row* [Function]

Returns the inverse of the `row` *row*. That is, the `row`, *r*, such that when *row* is permuted by *r*, the result is rounds. A theorem of group theory implies also that when *r* is permuted by *row* the result will also be rounds. Signals a `type-error` if *row* is not a `row`.

```
(inverse !13427586) ⇒ !14236857
(inverse !14236857) ⇒ !13427586
(inverse !12436587) ⇒ !12436587
(inverse !12345678) ⇒ !12345678
```

`permute-by-inverse` *row change* [Function]

Equivalent to `(permute` *row* `(inverse` *change*`))`. Signals a `type-error` if either *row* or *change* is not a `row`.

```
(permute-by-inverse !13456287 !45678123) ⇒ !28713456
(permute-by-inverse !54312 !2438756) ⇒ !54137862
(permute-by-inverse !762345 !4312) ⇒ !6271345
```

`alter-stage` *row* **&optional** *new-stage* [Function]

If there is a `row`, *r*, of stage *new-stage* such that `(equalp (permute (rounds` *new-stage*`)` *r*`)` *row*`)` then returns *r*, and otherwise `nil`. That is, it returns a row of the *new-stage* such that the first bells are as in *row*, and any new or omitted bells are in rounds order. If not supplied *new-stage* defaults to the current value of `*default-stage*`. Signals a `type-err` if *row* is not a `row` or *new-stage* is not a `stage`.

```
(alter-stage !54321 10) ⇒ !5432167890
(alter-stage !5432167890 6) ⇒ !543216
(alter-stage !54321 4) ⇒ nil
(alter-stage !5432167890 4) ⇒ nil
```

2.4 Place notation

Place notation is a succinct notation for writing sequences of changes, and is widely used in change ringing. Roan provides functions for reading and writing place notation, producing lists of `row`s, representing changes.

Place notation manipulated by Roan is extended to support jump changes and comma as an unfolding operator for easy notation of palindromic sequences of changes.

Jump changes may be included in the place notation in two ways. Within changes may appear parenthesized pairs of places, indicating that the bell in the first place jumps to the second place. Thus the change (13)6 corresponds to the jump change 231546. As usual implied leading or lying places may be omitted, so that could also be written simply (13). However, just as with ordinary place notation, all internal places must be noted explicitly; for example, the change (13)(31) is illegal, and must be written (13)2(31). Using this notation the first half-lead of London Treble Jump Minor can be written 3x3.(24)x2x(35).4x4.3.

Jump changes may also be written by writing the full row between square brackets. So that same half-lead of London Treble Jump Minor could instead be notated 3x3[134265]x2x[214536]4x4.3. Or they can be mixed 3x3[134265]x2x(35).4x4.3.

Palindromes may be conveniently notated using a comma operator, which means the changes preceding the comma are rung backwads, following the last of the changes before the comma, which is not repeated; followed by the changes following the comma, similarly unfolded. Thus x3x4,2x3 is equivalent to x3x4x3x2x3x2. A piece of place notation may include at most one comma. Neither the changes before the comma nor after it may be empty. Any piece of place notation including a comma is necessarily of even length.

If jump changes appear in place notation that is being unfolded then when rung in reverse the jump changes are inverted; this makes no difference to ordinary changes, which are always involutions, but is important for jump changes that are not involutions. If the central change about which the unfolding operation takes place, that is the last change in a sequence of changes being unfolded, is not an involution an error is signaled. As an example, a plain lead of London Treble Jump Minor can be notated as 3x3.(24)x2x(35).4x4.3,2 which is equivalent to 3x3.(24)x2x(35).4x4.3.4x4.(53)x2x(42).3x3.2.

While place notation is normally written using dots (full stops) only between non-cross changes, `parse-place-notation` will accept, and ignore, them between any changes, adjacent to other dots, and before and after place notation to be parsed. This may simplify operation with other software that emits place notation with extraneous dots.

Just as Roan augments the Lisp reader with '`!`' to read `row`s, it augments it with the '`#!`' reader macro to read place notatation. The stage at which the place notation is to be interpreted can be written as an integer between the '`#`' and the '`!`'. If no explict stage is provided the current value (at read time) of `*default-stage*` is used. The sequence of place notation must be followed by a character that cannot appear in place notation, such as whitespace, or by end of file. There is an exception that an unbalanced close parenthesis will also end the reading; this allows using this to read place notation in lists and vectors without requiring whitespace following the place notation. The place notation may be extended with the comma unfolding operator, and with jump changes. The stage at which the place notation is being iterpreted is not considered in deciding which characters to consume; all that might apply as place notation at any stage will be consumed. If some are not appropriate an error will only be signaled after all the continguous, place notation characters have been read.

Note that, unlike `row`s, which are Lisp atoms, the result of reading place notation is a list, so '`#!`' quotes it. This is appropriate in the usual case where the result of '`#!`' is evaluated, but if used in a context where it is not evaluated care must be exercised.

```
ROAN> #6!x2,1
(!214365 !124365 !214365 !132546)
ROAN> '(symbol #6!x2,1 x #6!x2x1)
(SYMBOL '(!214365 !124365 !214365 !132546) X
        '(!214365 !124365 !214365 !132546))
ROAN> `(symbol ,#6!x2,1 x ,#6!x2x1)
(SYMBOL (!214365 !124365 !214365 !132546) X
        (!214365 !124365 !214365 !132546))
ROAN> #6!x2
(!214365 !124365)
ROAN> (equalp #10!x1x4,2 #10!x1x4x1x2)
T
ROAN> #6!x3.(13)(64)
(!214365 !213546 !231645)
ROAN> #6!x3.(13).(64)
(!214365 !213546 !231546 !132645)
ROAN> #6!x3[231546](64)
(!214365 !213546 !231546 !132645)
```

`parse-place-notation` *string* **&key** *stage start end junk-allowed* [Function]

Parses place notation from *string*, returning a list of `row`s, representing changes, of stage *stage*. The place notation is parsed as applying to stage *stage*, which, if not supplied, defaults to current value of `*default-stage*`. Only that portion of *string* between *start* and *end* is parsed; *start* should be a non-negative integer, and *end* either an integer larger than *start* or `nil`, which latter is equivalent to the length of *string*. If *junk-allowed*, a generalized Boolean, is `nil`, the default, *string* must consist of the place notation parsed and nothing else; otherwise non-place notation characters may

follow the place notation. For purposes of parsing *stage* is not initially considered: if the place notation is only appropriate for higher stages it will not terminate the parse even if *junk-allowed* is true, it will instead signal an error. Two values are returned. The first is a list of `row`s, the changes parsed. The second is the index of the next character in *string* following the place notation that was parsed.

If the section of *string* delimited by *start* and *end* does not contain place notation suitable for *stage* a `parse-error` is signaled. If *string* is not a string, *stage* is not a `stage` or *start* or *end* are not suitable bounding index designators a `type-error` is signaled.

```
(multiple-value-list (parse-place-notation "x2.3" :stage 6))
    ⇒ ((!214365 !124365 !213546) 4)
```

`read-place-notation` **&optional** *stream stage eof-error-p eof-value recursive-p* [Function]

Reads place notation from a stream, resulting in a list of `row`s representing changes. Reads all the consecutive characters that can appear in (extended) place notation, and then tries to parse them as place notation. It accumulates characters that could appear as place notation at any stage, even stages above *stage*. The sequence of place notation must be followed by a character that cannot appear in place notation, such as whitespace, or by end of file. There is an exception, in that an unbalanced close parenthesis will also end the read; this allows using this to read place notation in lists and vectors without requiring whitespace following the place notation. The place notation may be extended with the comma unfolding operator, and with jump changes, as in `parse-place-notation`. The argument *stream* is a character stream open for reading, and defaults to the current value of `*standard-input*`; *stage* is a `stage`, an integer, and defaults to the current value of `*default-stage*`; and *eof-error-p*, *eof-value* and *recursive-p* are as for the standard `read` function, defaulting to `t`, `nil` and `nil`, respectively. Returns a non-empty list of `row`s, all of stage *stage*. Signals an error if no place notation constituents are available, if the characters read cannot be parsed as (extended) place noation at *stage*, or if one of the usual errorneous conditions while reading occurs.

`write-place-notation` *changes* **&key** *stream escape comma elide cross upper-case jump-changes* [Function]

Writes to *stream* characters representing place notation for *changes*, a list of `row`s.

The list *changes* should be a non-empty list of `row`s, all of the same stage. The *stream* should a character stream open for writing. It defaults to the current value of `*standard-output*`. If the generalized boolean *escape*, which defaults to the current value of `*print-escape*`, is true the place notation will be written using the '`#!`' read macro to allow the Lisp `read` function to read it; in this case the stage will always be explicitly noted between the '`#`' and the '`!`'. If the generalized boolean *upper-case*, which defaults to the current value of `*print-bells-upper-case*`, is true positions notated using letters will be written in upper case, and otherwise in lower case.

The argument *cross* controls which character is used to denote a cross change at even stages. It must be a character designator for `#\x`, `#\X` or `#\-`, and defaults to the current value of `*cross-character*`.

The argument *jump-changes* should be one of `nil`, `:jumps` or `:full`. It determines how jump changes will be notated. If it is `nil` and *changes* contains any jump changes an error will be signaled. If it is `:jumps` any jump changes will be notated using pairs of places between parentheses. While `parse-place-notation` and `read-place-notation` can interpret ordinary conjunct motion or even place making notated in parentheses, `write-place-notation` will only use parentheses for bells actually moving more than one place. If *jump-changes* is `:full` jump changes will be notated as a row between square brackets. Again, while ordinary changes notated this way can be parsed or read, `write-place-notation` will only use bracket notation for jump changes.

The argument *elide* determines whether, and how, to omit leading and/or lying places. If the stage of the changes in *changes* is odd, or if *elide* is `nil`, no such elision takes place. Otherwise *elide* should be one of `:interior`, `:leading`, `:lying` or `:lead-end`, which last is its default value. For any of these non-nil values leading or lying places will always be elided if there are interior places. They differ only for hunts (that is, changes with both a leading and lying place, and no interior places). If `:interior`, no elision takes place if there are no interior places. If `:leading`, the '1' is elided as implicitly available. If `:lying`, the lying place is elided, so that the result is always '1'. The value `:lead-end` specifies the same behavior as `:lying` for all the elements of *changes* except the last, for which it behaves as `:leading`; this is often convenient for notating leads of treble dominated methods at even stages.

If the generalized boolean *comma* is true an attempt is made to write *changes* using a comma operator separating it into palindromes. In general there can be multiple ways of splitting an arbitrary piece of place notation into palindromes. If this is the case the choice is made to favor first a division that has the palindrome after the comma of length one, and if that is not possible the division that has the shortest palindrome before the comma. Any sequence of changes of length two can be trivially divided into palindromes, but notating them with a comma is unhelpful, so *comma* applies only to even length lists of changes of length greater than two. Whether or not a partitioning into palindromes was possible can be determined by examining the second value returned by this function, which will be true only if a comma was written.

Returns two values, *changes*, and a generalized Boolean indicating whether or not the result was written with a comma.

Signals an error if *changes* is empty, or contains rows of different stages, if *stream* is not a character stream open for writing, or if any of the usual IO errors occurs.

`place-notation-string` *changes* **&key** *comma elide cross upper-case allow-jump-changes* [Function]

Returns a string of the place notation representing the list *changes*. The arguments are the same as the like named arguments to `write-place-notation`. A leading '#!' is never included in the result.

Signals a `type-error` if any elements of *changes* are not `rows`. Signals an error if *changes* is empty or contains rows of different stages.

```
(multiple-value-list
  (place-notation-string #8!x1x4,1 :elide nil))
    ⇒ ("x18x14x18x18" nil)
(multiple-value-list
  (place-notation-string #8!x1x4,1 :comma t))
    ⇒ ("x1x4,8" t)
(multiple-value-list
  (place-notation-string #8!x1x4,2 :elide :interior))
    ⇒ ("x18x4x18x18" nil)
```

`canonicalize-place-notation` *string-or-changes* **&key** *stage comma elide cross upper-case allow-jump-changes* [Function]

Returns a string representing the place notation in a canonical form. If *string-or-changes* is a string it should be parseable as place notation at *stage*, which defaults to the current value of `*default-stage*`, and otherwise it should be a list of `rows`, all of the same stage. Unless overridden by the other keyword arguments, which have the same effects as for `write-place-notation`, the canonical form is a compact one usin lower case '`x`' for cross, upper case letters for place high place names, `lead-end` style elision of external places, a comma for unfolding if possible, and notating jump changes as jumps within parentheses.

Signals a `type-error` if *string-or-changes* is neither a string nor a list, or if it is a list containing anything other than `rows`. Signals a `parse-error` if *string-or-changes* is a string and is not parseable at *stage*, or if *stage* is not a `stage`. Signals an error if *cross* is not a suitable character designator, if *allow-jump-changes* is not one of its allowed values, or if *string-or-changes* is a list containing `rows` of different stages. See [write-place-notation], page 16.

```
(multiple-value-list
  (canonicalize-place-notation "-16.X.14-6X1" :stage 6))
    ⇒ ("x1x4,6" t)
(multiple-value-list
  (canonicalize-place-notation "-3-[134265]-1T-" :stage 12))
    ⇒ ("x3x(24)x1x" nil)
```

`*cross-character*` [Variable]

The character used by default as "cross" when writing place notation. Must be a character designator for one of `#\x`, `#\X` or `#\-`. Its initial default value is a lower case '`x`', `#\x`.

3 Hash-sets

For change ringing applications it is often useful to manipulate sets of rows. That is, unordered collections of rows without duplicates. To support this and similar uses Roan supplies `hash-set`s, which use `equalp` as the comparison for whether or not two candidate elements are "the same". In addition, `equalp` can be used to compare two `hash-set`s themselves for equality: they are `equalp` if they contain the same number of elements, and each of the elements of one is `equalp` to an element of the other.

```
(equalp (hash-set !12345678 !13572468 !12753468 !13572468)
        (hash-set-union (hash-set !12753468 !12345678)
                        (hash-set !13572468 !12753468 !13572468)))
    ⇒ t
```

`hash-set` [Type]

A set data structure, with element equality determined by `equalp`. That is, no two elements of such a set will ever be `equalp`, only one of those added remaining present in the set. Set membership testing, adding new elements to the set, and deletion of elements from the set is, on average, constant time. Two `hash-set`s can be compared with `equalp`: they are considered `equalp` if and only if they contain the same number of elements, and each of the elements of one is `equalp` to an element of the other.

`make-hash-set` **&key** *size rehash-size rehash-threshold initial-elements* [Function]

Returns a new `hash-set`. If *initial-elements* is supplied and non-nil, it must be a list of elements that the return value will contain; otherwise an empty set is returned. If any of *size*, *rehash-size* or *rehash-threshold* are supplied they have meanings analagous to the eponymous arguments to `make-hash-table`.

`hash-set` **&rest** *initial-elements* [Function]

Returns a new `hash-set` containing the elements of *initial-elements*. If no *initial-elements* are supplied, the returned `hash-set` is empty.

```
(hash-set 1 :foo 2 :foo 1) ⇒ #<HASH-SET 3>
(hash-set-elements (hash-set 1 :foo 2 :foo 1))
    ⇒ (1 2 :foo)
(hash-set-elements (hash-set)) ⇒ nil
```

`hash-set-copy` *set* **&key** *size rehash-size rehash-threshold* [Function]

Returns a new `hash-set` containing the same elements as the `hash-set` *set*. If any of *size*, *rehash-size* or *rehash-threshold* are supplied they have the same meanings as the eponymous arguments to `copy-hash-table`. A `type-error` is signaled if *set* is not a `hash-set`.

3.1 Properties of hash-sets

`hash-set-count` *set* [Function]

Returns a non-negative integer, the number of elements the `hash-set` *set* contains. Signals a `type-error` if *set* is not a `hash-set`.

```
(hash-set-count (hash-set !1234 !1342 !1234)) ⇒ 2
(hash-set-count (hash-set)) ⇒ 0
```

`hash-set-empty-p` *set* [Function]
: True if and only if the `hash-set` *set* contains no elements. Signals a `type-error` if *set* is not a `hash-set`.

`hash-set-elements` *set* [Function]
: Returns a list of all the elements of the `hash-set` *set*. The order of the elements in the list is undefined, and may vary between two invocations of `hash-set-elements`. Signals a `type-error` if *set* is not a `hash-set`.

```
(hash-set-elements (hash-set 1 2 1 3 1)) ⇒ (3 2 1)
```

`hash-set-member` *item set* [Function]
: True if and only if *item* is an element of the `hash-set` *set*. Signals a `type-error` if *set* is not a `hash-set`.

```
(hash-set-member !1342 (hash-set !1243 !1342)) ⇒ t
(hash-set-member !1342 (hash-set !12435 !12425)) ⇒ nil
```

`hash-set-subset-p` *subset superset* [Function]
`hash-set-proper-subset-p` *subset superset* [Function]
: The `hash-set-subset-p` predicate is true if and only if all elements of *subset* occur in *superset*. The `hash-set-proper-subset-p` predicate is true if and only that is the case and further that `subset` does not contain all the elements of *superset*. `type-error` is signaled if either argument is not a `hash-set`.

```
(hash-set-subset-p (hash-set 1) (hash-set 2 1) ⇒ t
(hash-set-proper-subset-p (hash-set 1) (hash-set 2 1) ⇒ t
(hash-set-subset-p (hash-set 1 2) (hash-set 2 1) ⇒ t
(hash-set-proper-subset-p (hash-set 1 2) (hash-set 2 1) ⇒ nil
(hash-set-subset-p (hash-set 1 3) (hash-set 2 1) ⇒ nil
(hash-set-proper-subset-p (hash-set 1 3) (hash-set 2 1) ⇒ nil
```

3.2 Modifying hash-sets

`hash-set-clear` *set* [Function]
: Removes all elements from *set*, and then returns the now empty `hash-set`. Signals a `type-error` if *set* is not a `hash-set`.

`hash-set-adjoin` *set* **&rest** *elements* [Function]
`hash-set-nadjoin` *set* **&rest** *elements* [Function]
: Returns a `hash-set` contains all the elements of *set* to which have been added the *elements*. As usual duplicate elements are not added, though exactly which of any potential duplicates are retained is undefined. The `hash-set-adjoin` function returns a freshly created `hash-set` and does not modify *set*, while `hash-set-nadjoin` modifies and returns *set*. Signals a `type-error` if *set* is not a `hash-set`.

```
(hash-set-elements (hash-set-adjoin (hash-set 1 2 3) 4 3 2))
    ⇒ (3 4 1 2)
```

`hash-set-nadjoinf` *set* **&rest** *elements* [Macro]
: Adds *elements* to *set*, which should be a location suitable as a first argument to `setf` containing a `hash-set`, which is modified. As usual duplicate elements are not added,

though exactly which of any potential duplicates are retained is undefined. Returns *set* Signals a `type-error` if *set* does not contain a `hash-set`.

```
(let ((s (hash-set !1324 !3412 !4321)))
  (adjoinf s !1234 !3412 !4231)
  (hash-set-elements s))
    ⇒ (!3412 !4231 !1234 !3412 !4321 !1324)
```

`hash-set-remove` *set* **&rest** *elements* [Function]
Returns a new `hash-set` that contains all the elements of *set* that are not `equalp` to any of the *elements*. Signals a `type-error` if *set* is not a `hash-set`.

`hash-set-delete` *set* **&rest** *elements* [Function]
Deletes from the `hash-set` *set* all elements `equalp` to elements of *elements*, and returns the modified set. Signals a `type-error` if *set* is not a `hash-set`.

`hash-set-deletef` *set* **&rest** *elements* [Macro]
Deletes from *set*, which should be a location suitable as a first argument to `setf` contains a `hash-set`, all its elements `equalp` to any of the *elements*. Returns *set*. Signals a `type-error` if the *set* does not contain a `hash-set`.

```
(let ((s (hash-set !3524 !5432 !4253 !2345)))
  (hash-set-deletef s !2345 !5432)
  (hash-set-elements s))
   ⇒ (!4253 !3524)
```

`hash-set-difference` *set* **&rest** *more-sets* [Function]
`hash-set-ndifference` *set* **&rest** *more-sets* [Function]
Returns a `hash-set` containing all the elements of *set* that not contained in any of *more-sets*. The `hash-set-difference` version returns a fresh `hash-set`, and does not modify *set* or any of the *more-sets*. The `hash-set-ndifference` version modifies and returns *set*, but does not modify any of *more-sets*. Signals a `type-error` if *set* or any of *more-sets* are not `hash-set`s.

```
(hash-set-elements
  (hash-set-difference
    (hash-set !12345 !23451 !34512 !45123)
    (hash-set !23451 !54321 !12345)))
    ⇒ (!34512 !45123)
```

`hash-set-union` **&rest** *sets* [Function]
`hash-set-nunion` **&rest** *sets* [Function]
Returns a `hash-set` containing all the elements that appear in one or more of the *sets*. The `hash-set-union` version returns a fresh `hash-set`, and does not modify any of the *sets*. The `hash-set-nunion` may modify or destroy one or more of the *sets*, and the return value may or may not be `eq` to one of them. Signals a `type-error` if any of the *sets* are not `hash-set`s.

```
(coerce
  (hash-set-elements
    (hash-set-union
      (apply #'hash-set (coerce "abcdef" 'list))
      (apply #'hash-set (coerce "ACEG" 'list))))
  'string)
    ⇒ "FaeGbcd"
(hash-set-empty-p (hash-set-union)) ⇒ t
```

`hash-set-intersection` *set* **&rest** *more-sets* [Function]
`hash-set-nintersection` *set* **&rest** *more-sets* [Function]

Retuns a `hash-set` such at all of its elements are also elements of *set* and of all the *more-sets*. The `hash-set-intersection` version returns a fresh `hash-set`, and does not modify *set* or any of the *more-sets*. The `hash-set-nintersection` version may modify or destroy *set* and one or more of the *more-sets*, and the return value may or may not be `eq` to one of them. Signals a `type-error` if *set* or any of *more-sets* are not `hash-set`s.

```
(coerce
  (hash-set-elements
    (hash-set-intersection
      (apply #'hash-set (coerce "abcdef" 'list))
      (apply #'hash-set (coerce "ACEG" 'list))))
  'string)
    ⇒ "EaC"
```

3.3 Iterating over `hash-sets`

`map-hash-set` *function set* [Function]

Calls *function* on each element of the `hash-set` *set*, and returns `nil`. The order in which the elements of *set* have *function* applied to them is undefined. With one exception, the behavior is undefined if *function* attempts to modify the contents of *set*: *function* may call `hash-set-delete` to delete the current element, but no other. A `type-error` is signaled if *set* is not a `hash-set`.

```
(let ((r nil))
  (map-hash-set #'(lambda (e)
                    (push (list e (in-course-p e)) r))
                (hash-set !135246 !123456 !531246))
  r)
    ⇒ ((!135246 nil) (!531246 nil) (!123456 t))
```

`do-hash-set` (*var set* **&optional** *result-form*) **&body** *body* [Macro]

Evaluates the *body*, an implicit `progn`, repeatedly with the symbol `var` bound to the elements of the `hash-set` *set*. Returns the result of evaluating *result-form*, which defaults to `nil`, after the last iteration. A value may be returned by using `return` or `return-from nil`, in which case *result-form* is not evaluated. The order in which the elements of *set* are bound to `var` for evaluating *body* is undefined. With one exception the behavior is undefined if *body* attempts to modify the contents of *set*:

function may call `hash-set-delete` to delete the current element, but no other. A `type-error` is signaled if *set* is not a `hash-set`.

```
(let ((r nil))
  (do-hash-set (e (hash-set !135246 !123456 !531246) r)
    (push (list e (in-course-p e) r))))
    ⇒ ((!531246 nil) (!123456 t) (!135246 nil))
```

In addition, it is possible to iterate over a `hash-set` using the `iterate` (`https://common-lisp.net/project/iterate/`) macro, by using the `for...:in-hash-set...` construct.

```
(iter (for element :in-hash-set (hash-set !135246 !123456 !531246))
      (collect (list element (in-course-p element))))
    ⇒ ((!531246 nil) (!135246 nil) (!123456 t))
```

4 Patterns

Roan provides a simple pattern language for matching rows. This is useful, among other things, for counting rows considered particularly musical or unmusical.

A pattern string describes the bells in a row, with several kinds of wildcards and other constructs matching multiple bells. Bells' names match themselves, so, for example, `"13572468"` matches queens on eight. A question mark matches any bell, and an asterisk matches runs of zero or more bells. Thus `"*7468"`, at major, matches all twenty-four 7468s, and `"?5?6?7?8"` matches all twenty-four major rows that have the 5-6-7-8 in the positions they are in in tittums. Alternatives can be separated by the pipe character, '`|`'. Thus `"13572468|12753468"` matches either queens or Whittingtons. Concatentation of characters binds more tightly than alternation, but parentheses can be used to group subexpressions. Thus `"*(4|5|6)(4|5|6)78"` at major matches all 144 combination rollups. When matched against two major rows `"?*12345678*?"` matches wraps of rounds, but not either row being rounds.

Two further notations are possible. In each case it does not extend what can be expressed, it merely makes more compact something that can be expressed with the symbols already described. The first is a bell class, which consits of one or more bell names within square brackets, and indicates any one of those bells. Thus an alternative way to match the 144 combination rollups at major is `"*[456][456]78"`.

A more compact notation is also available for describing runs of consecutive bells. Two bell symbols separated by a hyphen represent the run of bells from one to the other. Thus `"*5-T"` matches all rows ending 567890ET. If such a run description is followed by a solidus, '`/`', and a one or two digit integer, it matches all runs of the length of that integer that are subsequences of the given run. Thus `"*2-8/4"` is equivalent to `"*(2345|3456|4567|5678)"`. If instead of a solidus a percent sign, '%', is used it matches subsequences of both the run and its reverse. Thus `"1-6%4*"` matches all little bell runs off the front of length four selected from the bells 1 through 6, and is equivalent to the pattern `"(1234|4321|2345|5432|3456|6543)*"`. There is some possible ambiguity with this notation, in that the second digit of an integer following a solidus or percent sign could be interpreted as a digit or a bell symbol. In these cases it is always interpreted as a digit, but the other use can be specified by using parentheses or a space.

Spaces, but no other whitespace, can be included in patterns. However no spaces may be included within bell classes or run descriptions. Thus `" 123 [456] 7-T/3 * "` is equivalent to `"123[456]7-T/3*"`, but both `"123[ 4 5 6 ]7-T/3*"` and `"123[456]7-T / 3*"` are illegal, and will cause an error to be signaled.

In addition to strings, patterns may be represented by parse trees, which are simple list structures made up of keywords and bells (that is, small, non-negative integers). Strings are generally more convenient for reading and writing patterns by humans, but parse trees can be more convenient for programmatically generated patterns. The function `pattern-parse` converts the string representation of a pattern to such a tree structure. Sequences of elements are represented by lists starting with `:sequence`; alternatives by lists starting with `:or`; bell classes by lists of the included bells preceded by `:class`; runs by a list of the form `(:run` *`start end length bi`*`)`, where *start* is the starting `bell`, *end* the ending `bell`, *length* the length of the run, and *bi* is a generalized boolean saying whether or not the runs are bidirectional; `bell`s are represented by themselves; and '`?`' and '`*`' by `:one` and `:any`,

respectively. The elements of the `:sequence` and `:or` lists may also be lists themselves, representating subexpressions. For example, the string `"(?[234]*|*4-9%4?)*T"` is equivalent to the tree

```
(:sequence (:or (:sequence :one (:class 1 2 3) :any)
                (:sequence :any (:run 3 8 4 t) :one))
           :any
           11)
```

`row-match-p` *pattern row* **&optional** *following-row* [Function]

Determines whether *row*, or pair of consecutive `row`s, *row* and *following-row*, match a pattern. If *following-row* is supplied it should be of the same stage as *row*. The *pattern* may be a string or a tree, and should be constructed to be appropriate for the stage of *row*; an error is signaled if it contains explicit matches for bells of higher stage than *row*. Returns a generalized boolean indicating whether or not *pattern* matches.

```
(row-match-p "*[456][456]78" !32516478) ⇒ t
(row-match-p "*[456][456]78" !12453678) ⇒ nil
(row-match-p "*[456][456]78" !9012345678) ⇒ t
(row-match-p "?*123456*?" !651234 !562143) ⇒ t
(row-match-p "?*123456*?" !651234 !652143) ⇒ nil
(row-match-p "?*123456*?" !123456) ⇒ nil
(row-match-p '(:sequence :any 6 7) !65432178) ⇒ t
(row-match-p '(:sequence :any 6 7) !23456781) ⇒ nil
```

Signals an error if *pattern* cannot be parsed as a pattern, if *row* is not a `row`, if *following-row* is neither a `row` nor `nil`, if *pattern* contains bells above the stage of *row*, or if *following-row* is a `row` of a different stage than *row*.

Care should be used when matching against two rows. In the usual use case when searching for things like wraps every row typically will be passed twice to this method, first as *row* and then as *following-row*. A naive pattern might end up matching twice, and thus double counting. For example, if at major `"*12345678*"` were used to search for wraps of rounds it would match whenever *row* or *following-row* were themselves rounds, possibly leading to double counting. Instead a search for wraps of rounds might be better done against something like `"?*12345678*?"`.

`parse-pattern` *pattern* **&optional** *stage* [Function]

Converts a string representation of a pattern to its parse tree, and returns it. The *stage* is the stage for which *pattern* is parsed, and defaults to `*default-stage*`. If *pattern* is a non-empty list it is presumed to be a pattern parse tree and is returned unchanged. Signals a `type-error` if *pattern* is neither a string nor a non-empty list, or if *stage* is not a `stage`. Signals a `parse-error` if *pattern* is a string but cannot be parsed as a pattern, or contains bells above those appropriate for *stage*.

```
(parse-pattern "(?[234]*|*4-9%4?)*T" 12)
    ⇒
  (:sequence (:or (:sequence :one (:class 1 2 3) :any)
                  (:sequence :any (:run 3 8 4 t) :one))
             :any
             11)
```

`format-pattern` *tree* **&optional** *upper-case* [Function]

Returns a string that if parsed with `parse-pattern`, would return the parse tree *tree*. Note that the generation of a suitable string from *tree* is not unique, and this function simply returns one of potentially many equivalent possibilities. The case of any bells represented by letters is controlled by *upper-case*, which defaults to the current value of `*print-bells-upper-case*`. Signals an error if tree is not a parse tree for a pattern.

```
(format-pattern '(:sequence 0 1 2 :any 7) t) ⇒ "123*8"
```

`named-row-pattern` *name* **&optional** *stage covered* [Function]

Returns a pattern, as a parse tree, that matches a named row at *stage*. The *name* is one of those listed below. If *stage* is not supplied it defaults to the current value of `*default-stage*`. If *covered*, a generalized boolean, is non-nil the `row`('s) that will be matched will assume an implicit tenor. If *covered* is not supplied it defaults to `nil` for even stages and `t` for odd stages. If there is no such named row known that corresponds to the values of *stage* and *covered* `nil` is returned. Signals an error if *name* is not a keyword or is not a known named row name as enumerated below, or if *stage* is not a `stage`.

The supported values for *name*, and the stages at which they are defined, are:

`:backrounds`
: any stage

`:queens`
: uncovered singles and above, or covered two and above.

`:kings`
: uncovered minimus and above, or covered singles and above; note that kings at uncovered minor or covered doubles is the same row as Whittingtons at those stages

`:whittingtons`
: uncovered minor and above, or covered doubles and above; note that Whittingtons at uncovered minor or covered doubles is the same row as kings at those stages

`:double-whittingtons`
: covered cinques or uncovered maximus, only

`:roller-coaster`
: covered caters or uncovered royal, only

`:near-miss`
: any stage

```
(format-pattern (named-row-pattern :whittingtons 10 nil))
    ⇒ "1234975680"
(format-pattern (named-row-pattern :whittingtons 9 t)
    ⇒ "123497568"
(format-pattern (named-row-pattern :whittingtons 9 nil))
    ⇒ "123864579"
(named-row-pattern :whittingtons 4)
    ⇒ nil
```

`pattern-parse-error` [Type]

An error signaled when attempting to parse a malformed row pattern. Contains three potenitally useful slots accessible with `pattern-parse-error-message`, `pattern-parse-error-pattern` and `pattern-parse-error-index`.

4.1 Counting matches

Often one would like to count how many times a variety of patterns match many different rows. To support this use Roan provides `match-counter`s. After creating a `match-counter` with `make-match-counter` you add a variety of patterns to it, with `add-pattern` or `add-patterns`, each with a label, which will typically be a symbol or string, but can be any Lisp object. You then apply the `match-counter` to `row`s with `record-matches`, and query how many matches have occurred with `match-counter-counts`.

The order in which patterns are added to a `match-counter` is preserved, and is reflected in the return values of `match-counter-labels`, and `match-counter-counts` called without a second argument. Replacing an existing pattern by adding a different one with a *label* that is `equalp` to an existing one does not change the order, but deleting a pattern with `remove-pattern` and then re-adding it does move it to the end of the order. When a pattern has been replaced by one with an `equalp` *label* that is not `eq` to the original *label* which label is retained is undefined.

A `match-counter` also distinguishes matches that occur at handstroke from those that occur at backstroke. Typically you tell the `match-counter` which stroke the next `row` it is asked to match is on, and it then automatically alternates handstrokes and backstrokes for subsequent `row`s. For patterns that span two rows, such as wraps, the stroke is considered to be that between the rows; for example a wrap of rounds that spans a backstroke lead would be considered to be "at" backstroke.

```
(let ((m (make-match-counter 8)))
  (add-patterns m '((cru "*[456][456]78")
                    (wrap "?*12345678*?" t)
                    (lb4 "1-7%4*|*1-7%4")))
  (loop for (row following)
        on (generate-rows #8!36.6.5.3x5.56.5,2)
        do (record-matches m row following))
  (values (match-counter-counts m)))
    ⇒ ((cru . 3) (wrap . 1) (lb4 . 5))
```

`match-counter` [Type]

Used to collect statistics on how many rows match a variety of patterns.

`make-match-counter` **&optional** *stage* [Function]

Returns a fresh `match-counter`, initially containing no patterns, that is configured to attempt to match patterns against `row`s of *stage* bells. If not supplied, *stage* defaults to the current value of `*default-stage*`. Attempts to add patterns only appropriate for a different stage or match rows of a different stage with `record-matches` will signal an error.

`add-pattern` *counter label pattern* **&optional** *double-row-p* [Function]
`add-patterns` *counter lists* [Function]

Adds one or more patterns to those matched by the `match-counter` *count*. A single pattern, *pattern*, is added, with label *label*, by `add-pattern`. If the generalized boolean *double-row-p* is true two rows (which typically should be consecutive) will be matched against *pattern*, and others one row; if not supplied *double-row-p* is `nil`. Multiple patterns may be added together with `add-patterns`: *lists* should be a list of lists, where the sublists are of the form `(`*`label pattern`* `&optional` *`double-row-p`*`)`, and the patterns are added in the order given. In either case the *pattern* may be either a string or list structure that is a parsed pattern, such as returned by `parse-pattern`. If *label* is `equalp` to the label of a pattern already added to *counter* that pattern will be replaced, and its corresponding counts reset to zero. Either function reeturns *counter*. Either signals a `type-error` if *counter* is not a `match-counter`. Signals an error if any of the *pattern*s are not an appropriate pattern for the stage of *counter*.

`remove-pattern` *counter label* [Function]

Removes any pattern in `method-counter` *count* with its label `equalp` to *label*. Returns `t` if such a pattern was found and removed, and `nil` otherwise. Signals a `type-error` if *count* is not a `method-counter`.

`remove-all-patterns` *counter* [Function]

Removes all the patterns in the `method-counter` *counter*, and returns a positive integer, the number of patterns so removed, if any, or `nil` if *counter* had no patterns. Signals a `type-error` if *counter* is not a `match-counter`.

`match-counter-pattern` *counter label* **&optional** *as-string* [Function]
upper-case

Returns two values: the first is the pattern whose label in *count* is `equalp` to *label*, if any, and otherwise `nil`; the second is a generalized boolean if and only if the first value is non-nil and the pattern is to be matched against two rows rather than just one. If the generalized boolean *as-string* is true the pattern is returned as a string, as by `format-pattern`, with the case of any bells represented by letters controled by the generalized boolean *upper-case*; and otherwise as a parse tree, as by `parse-pattern`. A string return value may not be `string-equal` to that added to *counter*, but will match the same `row`s. If *as-string* is not supplied it defaults to true; if *upper-case* is not supplied it defaults to the current value of `*print-bells-upper-case*`. Signals a `type-error` if *counter* is not a `match-counter`.

`match-counter-labels` *counter* [Function]

Returns two lists, the labels of those patterns in *count* that are matched against a single row, and those that are matched against two rows. Both lists are in the order in which the corresponding patterns were first added to *counter*. Signals a `type-error` if *counter* is not a `match-counter`.

`match-counter-counts` *counter* **&optional** *label* [Function]

Returns three values, the number of times the pattern with label `equalp` to *label* in *counter* has matched `row`s presented to it with `record-matches` since *counter* was reset or the relevent pattern was added to it. The first return value is the total number of matches, the second the number of matches at handstroke, and the third

the number of matches at backstroke. If no *label* is supplied it instead returns three a-lists mapping the labels of the patterns in *counter* to the number of matches, again total, handstroke and backstroke. The elements of these a-lists are in the order in which the corresponding patterns were first added to *counter*. Returns `nil` if there is no pattern labeled *label*. Signals a `type-error` if *counter* is not a `match-counter`.

`reset-match-counter` *counter* [Function]
Resets all the counts associated with all the patterns in *counter* to zero. Signals a `type-error` if *counter* is not a `match-counter`.

`match-counter-handstroke-p` *counter* [Function]
Returns a generalized boolean indicating that the next row presented to *counter* will be a handstroke. Can be used with `setf` to tell *counter* whether or not it should consider the next row a handstroke or a backstroke. If not explicitly set again, either with `(setf match-counter-handstroke-p)`, or with the *handstroke-p* argument to `record-matches`, whether or not subsequent rows will be considered handstroke or backstroke will alternate. Signals a `type-error` if *counter* is not a `match-counter`.

`record-matches` *counter row* **&optional** *following-row handstroke-p* [Function]
Causes all the single-row patterns of *counter* to be matched against *row*, and, if a *following-row* is supplied and not `nil`, also all the double-row patterns to be matched against both rows. If the generalized boolean *handstroke-p* is supplied it indicates whether *row* is to be considered a handstroke or not, and, unless explicitly set again, either with the *handstroke-p* argument to `record-matches` by with `(setf match-counter-handstroke-p)`, whether or not subsequent rows will be considered handroke or backstroke will alternate. That is, supplying a *handtroke-p* argument to `record-matches` is equivalent to calling `(setf match-counter-handstoke-p)` immediately before it. Signals a `type-error` if *counter* is not a `match-counter`, *row* is not a `row`, or *following-row* is neither a `row` nor `nil`.

5 Methods

Roan provides the `method` type to describe change ringing methods, not to be confused with CLOS methods. A `method` can only describe a method that can be viewed as a fixed sequence of changes, including jump changes; while this includes all methods recognized by the Central Council (as of mid-2017), and many others, it does exclude, for example, Dixonoids. A `method` has a stage, a name, an associated place-notation, and a property list, though any or all of these may be `nil`. In the case of the stage, name or place notation `nil` indicates that the corresponding value is not known. The stage, if known, should be a `stage`, and the name and place notation, if known, should be strings. The name is not just the portion that the Central Council considers its name: it also includes any explictly named class, such as 'Surprise', as well as modifiers such as 'Little' or 'Differential'. For example, the name of Littleport Little Surprise Royal is `"Littleport Little Surprise"`.

The property list may be used to store further information about a method. For example, `lookup-methods-by-name` typically adds the properties `:first-tower` and `:first-hand` to methods it creates with details of their first performances. Besides accessing the property list as a whole with `method-properties` individual properties can be interrogated and set with `method-property`.

Because a ringing method is unrelated to a CLOS method the `roan` package shadows `common-lisp:method` and `common-lisp:method-name`.

`method` [Type]

Describes a change ringing method, typically including its name, stage and place notation, and possibly other properties of the method.

`method` **&key** *title stage name place-notation properties* [Function]

Creates a new `method` instance, with the specified *stage*, *name* *place-notation* and *properties*. If *title* is supplied it provides default values for *name* and *stage* if none are explicitly provided or are `nil`. If *stage* is not provided, and no default is provided by *title*, it defaults to the current value of `*default-stage*`. A `type-error` is signaled if *stage* is supplied and is neither `nil` nor a `stage`; if any of *name*, *title* or *place-notation* are supplied and are neither `nil` nor a string; or if `properties` is supplied and is not a list.

`method-stage` *method* [Function]
`method-name` *method* [Function]
`method-place-notation` *method* [Function]
`method-properties` *method* [Function]

Returns the stage, name, place notation or property list of *method*, or `nil`. A non-nil value returned by `method-stage` is a `stage`, and by `method-name` or `method-place-notation` a string; `nil` for these indicates the corresponding attribute of *method* is unknown. The value returned by `method-properties` is a list, possibly empty. Note that the name is not just the portion that the Central Council considers its name: it also includes any explicitly named class, such as '`Surprise`', as well as modifiers such as '`Little`' or '`Differential`'. For example, the name of Littleport Little Surprise Royal is '`Littleport Little Surprise`'. All these functions may be used with `setf` to set the relevant attributes of *method*. No checking is done that the string

supplied as the `method-place-notation` is, in fact valid place notation; however, a subsequent attempt to use invalid place notation, for example by `method-changes` or `method-lead-head`, will signal an error. These functions all signal a `type-error` if *method* is not a `method`, as does attempting to set the stage to a non-`stage`, the name or place notation to a non-string, or the property list to a non-list.

`method-property` *method key* **&optional** *default* [Function]

Returns the property attached to *method* with the given *key*, if there is one, and otherwise returns *default*, or `nil` if no *default* is supplied. Can be used with `setf` to change a property. Signals a `type-error` if *method* is not a `method`. Just a short-hand for, and equivalent to, `(getf (method-properties method) key default)`.

```
(let ((m (method :title "Advent Surprise Major")))
  (setf (method-property m :first-rung) "1988-07-31")
  (method-property m :first-rung))
    ⇒ "1988-07-31"
```

`method-title` *method* **&optional** *show-unknown* [Function]

Returns a string containing as much of the *method*'s title as is known. If *show-unknown*, a generalized boolean defaulting to false, is not true then an unknown name is described as "Unknown", and otherwise is simply omitted. If neither the name nor the stage is known, and *show-unknown* is `nil`, then `nil` is returned instead of a string. Can be used with `setf`, in which case it potentially sets both the name and stage of *method*.

```
(method-title (method :stage 8 :name "Advent Surprise"))
  ⇒ "Advent Surprise Major"
(method-title (method :stage 8))
  ⇒ "Major"
(method-title (method :stage 8) t)
  ⇒ "Unknown Major
(method-title (method :stage nil :name "Advent Surprise"))
  ⇒ "Advent Surprise"
(method-title (method :stage nil))
  ⇒ nil
```

`parse-method-title` *string* [Function]

Returns two values, the method name and the stage extracted from *string*, if they are present. If one or both components is missing `nil` is returned as the corresponding value. Signals a `type-error` if *string* is neither a non-empty string nor `nil`.

```
(multiple-value-list
  (parse-method-title "Advent Surprise Major"))
    ⇒ ("Advent Surprise" 8)
```

`canonicalize-method-place-notation` *method* **&key** *comma elide cross upper-case allow-jump-changes* [Function]

Replaces *method*'s place-notation by an equivalent string in canonical form, and returns that canonical notation as a string. Unless overriden by keyword arguments this is a compact version with leading and lying changes elided according to `:lead-end`

format as for `write-place-notation`, partitioned with a comma, if possible, with upper case letters for high number bells and a lower case 'x' for cross. The behavior can be changed by passing keyword arguments as for `write-place-notation`. If *method* has no place-notation or no stage, this function does nothing, and returns `nil`; in particular, if there is place-notation but no stage, the place-notation will be unchanged.

Signals a `type-error` if *method* is not a `method`, and signals an error if any of the keyword arguments do not have suitable values for passing to `write-place-notation`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage. See [canonicalize-place-notation], page 18, and [write-place-notation], page 16.

```
(let ((m (method :stage 6
                 :place-notation "-16.X.14-6X16")))
  (canonicalize-method-place-notation m)
  (method-place-notation m))
    ⇒ "x1x4,6"
```

`method-changes` *method* [Function]

If *method*'s stage and place-notation have been set returns a fresh list of `rows`, representing changes, that constitute a plain lead of *method*, and otherwise returns `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-changes (method :stage 6
                        :place-notation "x2,6"))
    ⇒ (!214365 !124365 !214365 !132546)
```

`method-contains-jump-changes` *method* [Function]

If *method*'s stage and place-notation have been set and method contains one or more jump changes returns true, and otherwise returns `nil`. Note that even if the place notation is set and implies jump changes, if the stage is not set `method-contains-jump-changes` will still return `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-contains-jump-changes
  (method :place-notation "x3x4x2x3x4x5,2"
          :stage 6))
    ⇒ nil
(method-contains-jump-changes
  (method :place-notation "x3x(24)x2x(35)x4x5,2"
          :stage 6))
    ⇒ t
(method-contains-jump-changes
  (method :stage 6))
    ⇒ nil
(method-contains-jump-changes
  (method :place-notation "x3x(24)x2x(35)x4x5,2"
          :stage nil))
    ⇒ nil
```

`method-lead-head` *method* [Function]

If *method*'s stage and place-notation have been set returns a `row`, the lead head generated by one plain lead of *method*, and otherwise `nil`. If *method* has a one lead plain course the result will be rounds. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-lead-head (method :stage 8
                          :place-notation "x1x4,2"))
    ⇒ !16482735
```

`method-lead-head-code` *method* [Function]

Returns the lead head code for *method* if its stage and place notation are set and it has Plain Bob lead ends, and otherwise returns `nil`. Considers neither twin hunt methods, such as Grandsire, nor any method below minimus, as having Plain Bob lead ends, and Rounds is not considered a Plain Bob lead head. When not `nil` the result is a string containing a lower case letter, possibly followed by a digit.

The CCCBR's various collections of methods have, for several decades, use succinct codes to denote the combination of a Plain Bob lead head and the change that leads to it (`http://methods.org.uk/online/notes.htm`). While officially just a convention for these collections it has become a de facto standard. Unfortunately these codes are not precisely defined, and there are ambiguities for some uncommon hunt paths. Consequently this function only returns a non-`nil` result for single hunt, non-hybrid methods, including little methods, that do not contain any jump changes. Thus there are cases where a code is used in a CCCBR collection, but not returned by `method-lead-head-code`. It is also worth noting that for odd stages there are Plain Bob lead heads for which the CCCBR does not define any code; when called on such a method `method-lead-head-code` also returns `nil`.

Signals a `type-error` if *method* is not a `method`, and a `parse-error` if *method*'s place notation cannot be interpreted at its stage.

```
(method-lead-head-code
  (lookup-method "Ashtead Surprise" 8))
    ⇒ "d"
(method-lead-head-code
  (lookup-method "Zanussi Surprise" 12))
    ⇒ "j2"
(method-lead-head-code
    (lookup-method "Twerton Little Bob" 9))
      ⇒ "q1"
(method-lead-head-code
  (lookup-method "Double Glasgow Surprise" 8))
    ⇒ nil
```

`method-lead-count` *method* [Function]

If *method*'s stage and place-notation have been set returns a positive integer, the number of leads in a plain course of *method*, and otherwise `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-lead-count
  (method :title "Cambridge Surprise Minor"
          :place-notation "x3x4x2x3x4x5,2"))
    ⇒ 5
(method-lead-count
  (method :title "Cromwell Tower Block Minor"
          :place-notation "3x3.4x2x3x4x3,6"))
    ⇒ 1
(method-lead-count
  (method :title "Bexx Differential Bob Minor"
          :place-notation "x1x1x23,2"))
    ⇒ 6
```

`method-plain-lead` *method* [Function]

If *method*'s stage and place-notation have been set returns a fresh list of `row`s, starting with rounds, that constitute the first lead of the plain course of *method*, and otherwise returns `nil`. The lead head that starts the next lead is not included. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-plain-lead (method :stage 6
                           :place-notation "x2,6"))
    ⇒ (!123456 !214365 !213456 !124365)
```

`method-lead-length` *method* [Function]

If *method*'s stage and place-notation have been set returns a positive integer, the length of one lead of *method*, and otherwise `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-lead-length
  (method :title "Cambridge Surprise Minor"
          :place-notation "x3x4x2x3x4x5,2"))
    ⇒ 24
```

`method-course-length` *method* [Function]

If *method*'s stage and place-notation have been set returns a positive integer, the length of a plain course of *method*, and otherwise `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-course-length
  (method :title "Cambridge Surprise Minor"
          :place-notation "x3x4x2x3x4x5,2"))
    ⇒ 120
(method-course-length
  (method :title "Cromwell Tower Block Minor"
          :place-notation "3x3.4x2x3x4x3,6"))
    ⇒ 24
(method-course-length
  (method :title "Bexx Differential Bob Minor"
          :place-notation "x1x1x23,2"))
    ⇒ 72
```

`method-plain-course` *method* [Function]

If *method*'s stage and place-notation have been set returns a fresh list of the `row`s that constitute a plain course of *method*, and otherwise `nil`. The list returned will start with rounds, and end with the `row` immediately preceding the final rounds. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

`method-true-plain-course-p` *method* **&optional** [Function]
error-if-no-place-notation

If *method* has a non-nil stage and place notation set, returns true if *method*'s plain course is true and `nil` otherwise. If *method* does not have a non-nil stage or place notation a `no-place-notation-error` is signaled if the generalized boolean *error-if-no-place-notation* is true, and otherwise `nil` is returned; if *error-if-no-place-notation* is not supplied it defaults to true. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-true-plain-course-p
  (method :title "Little Bob Minor"
          :place-notation "x1x4,2"))
    ⇒ t
(method-true-plain-course-p
  (method :title "Unnamed Little Treble Place Minor"
          :place-notation "x5x4x2,2"))
    ⇒ nil
```

`method-hunt-bells` *method* [Function]
`method-principal-hunt-bells` *method* [Function]
`method-secondary-hunt-bells` *method* [Function]

If *method*'s stage and place-notation have been set `method-hunt-bells` returns a fresh list of `bell`s (that is, small integers, with the treble represented by zero) that are hunt bells of *method* (that is, that return to their starting place at each lead head), and otherwise returns `nil`. The bells in the list are ordered in increasing numeric order. Note that for a method with no hunt bells this function will also return `nil`.

The CCCBR's taxonomy of methods is largely driven by a division of hunt bells into "principal" hunt bells and "secondary" hunt bells (see the Central Council's Decisions (`http://www.methods.org.uk/ccdecs.htm`) for details). The `method-principal-hunt-bells` and `method-secondary-hunt-bells` functions return lists, again ordered in increasing numeric order, of these hunt bells. The lists returned by these two functions are disjoint, and, in the absence of jump changes, their union consists exactly of the contents of the list returned by `method-hunt-bells`. The CCCBR's taxonomy, and its definitions of principal and secondary hunt bells, is predicated on the absence of jump changes, and it is not clear how they would best be extended for methods with jump changes. The `method-principal-hunt-bells` and `method-secondary-hunt-bells` functions therefore return `nil` for any methods containing jump changes; however, `method-hunt-bells` can still be usefully used for such methods. Note that `method-principal-hunt-bells` and `method-secondary-hunt-bells` also return `nil` for methods without jump changes that do not contain any of the relevant hunt bells, as well as for methods that have not had both their stage and place-notation set.

All three of these functions ignal a `type-error` if *method* is not a `method`, and signal a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-hunt-bells (method :stage 5
                           :place-notation "3,1.5.1.5.1"))
    ⇒ (0 1)
(method-principal-hunt-bells (method :stage 5
                           :place-notation "3,1.5.1.5.1"))
    ⇒ (0 1)
(method-secondary-hunt-bells (method :stage 5
                           :place-notation "3,1.5.1.5.1"))
    ⇒ nil
(method-principal-hunt-bells (method :stage 5
                           :place-notation "5.1.5.1.125,2"))
    ⇒ (0)
(method-secondary-hunt-bells (method :stage 5
                           :place-notation "5.1.5.1.345,2"))
    ⇒ (1)
```

`method-working-bells` *method* [Function]

If *method*'s stage and place-notation have been set returns a list of lists of `bell`s (that is, small integers, with the treble represented by zero) that are working bells of *method*

(that is, that do not return to their starting place at each lead head), and otherwise returns `nil`. The sublists each represent a cycle of working bells. For example, for a major method with Plain Bob lead heads, there will be one sublist returned, of length seven, containing the bells 1 through 7; while for a differential method there will be at least two sublists returned. Each of the sublists is ordered starting with the smallest bell in that sublist, and then in the order the place bells follow one another in the method. Within the overall, top-level list the sublists are ordered such that the first element of each sublist occur in increasing numeric order. Note that for a method with no working bells (which will then have a one lead plain course) this function also returns `nil`. Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

```
(method-working-bells (method :stage 7
                              :place-notation "7.1.7.47,27"))
    ⇒ ((1 4 5) (2 6 3))
```

`method-rotations-p` *method-1 method-2* [Function]

Returns true if and only if the changes constituting a lead of *method-1* are the same as those constituting a lead of *method-2*, possibly rotated. If the changes are the same even without rotation that is considered a trivial rotation, and also returns true. Note that if *method-1* and *method-2* are of different stages the result will always be false.

Signals a `no-place-notation-error` if either argument does not have its stage or place notation set. Signals a `type-error` if either argument is not a `method`. Signals a `parse-error` if the place notation of either argument cannot be parsed as place notation at its stage.

```
(method-rotations-p
  (method :stage 5 :place-notation "3,1.5.1.5.1")
  (method :stage 5 :place-notation "5.1.5.1,1.3"))
    ⇒ t
(method-rotations-p
  (method :stage 5 :place-notation "3,1.5.1.5.1")
  (method :stage 5 :place-notation "3,1.5.1.5.1"))
    ⇒ t
(method-rotations-p
  (method :stage 5 :place-notation "3,1.5.1.5.1")
  (method :stage 5 :place-notation "3,1.3.1.5.1")
    ⇒ nil
(method-rotations-p
  (method :stage 5 :place-notation "3,1.5.1.5.1")
  (method :stage 7 :place-notation "5.1.5.1,1.3"))
    ⇒ nil)
```

`method-canonical-rotation-key` *method* [Function]

If *method* has its stage and place notation set returns a string uniquely identifying, using `equal`, the changes of a lead of this method, invariant under rotation. That is, if two methods are rotations of one another their `method-canonical-rotation-key`s will always be `equal`. The string is, other than being a string, essentially an

opaque type and should generally not be displayed to an end user or otherwise have its structure depended upon. Case is significant; `equalp` should not be used to compare these keys. While, within one version of Roan, this key can be counted on to be the same in different sessions and on different machines, it may change between versions of Roan. If *method* does not have both its stage and place notation set `method-canonical-rotation-key` returns `nil`.

Signals a `type-error` if *method* is not a `method`. Signals a `parse-error` if *method*'s place notation cannot be properly parsed at its stage.

```
(method-canonical-rotation-key
  (lookup-method "Cambridge Surprise" 8))
    ⇒ "bAvzluTjWO5P"
(method-canonical-rotation-key
  (method :stage 8 :place-notation "5x6x7,x4x36x25x4x3x2"))
    ⇒ "bAvzluTjWO5P"
(method-canonical-rotation-key
  (method :stage 8 :place-notation "x1x4,2"))
    ⇒ "bEvy3Zo"
(method-canonical-rotation-key
  (method :stage 10 :place-notation "x1x4,2"))
    ⇒ "Oi3Jd2sC"

(method-canonical-rotation-key (method) ⇒ nil
```

`method-classification` *method* [Function]

If *method* has had its stage and place-notation set returns a list of keywords describing how it fits into the CCCBR's taxonomy of methods (see Central Council's Decisions (`http://www.methods.org.uk/ccdecs.htm`)), and otherwise `nil`. In the absence of jump changes the first element of the result will be one of `:principle`, `:bob`, `:place`, `:slow-course`, `:treble-bob`, `:delight`, `:surprise`, `:treble-place`, `:alliance` or `:hybrid`. This may be followed by `:differential` and/or `:little`, if appropriate. While the CCCBR considers "differentials" as a distinct class from "principles", what the CCCBR calls a pure "differential" is here described as `(:principle :differential)`, in parallel with with "differential hunters[1]" Note that a principle, whether or not differential, can never be little.

The CCCBR's taxonomy deals only with methods not containing jump changes, and it is not at all clear how it should be extended to deal with methods that do contain them. Therefore `method-classificaiton` simply returns `(:jump)` for any such methods, and never appends `:differential` or `:little` to such a classification. Be careful when constructing the titles of such methods as the bands that have rung them have sometimes used more complex names, such as "treble jump" to describe them.

The CCCBR's taxonomy unfortunately does contain some ambiguities around unusual methods; for example little treble dodging methods containing multiple hunt bells with different cross section locations. The `method-classification` function will

[1] No, "differential hunter" is not a description of an infintesimal horse used for sport, it is an unattractive name for a meta-class of methods. See the CCCBR Decisions for further details.

take its best guess in such cases, but there is no guarantee that what it returns is how the Council will end up classifying such methods if they are rung and named.

Signals a `type-error` if *method* is not a `method`, and a `parse-error` if its place-notation cannot be parsed at its stage.

See also [set-method-classified-name], page 39, and [method-hunt-bells], page 36.

```
(method-classification (method :stage 8
                               :place-notation "x3x4x2x34,2")
  ⇒ (:surprise :little)
(method-classification (method :stage 11
                               :place-notation "3.1.E.3.1.3,1")
  ⇒ (:principle)
(method-classification (method :stage 12
                               :place-notation "58x1x67x67x49x,x")
  ⇒ (:hybrid :differential :little)
```

`set-method-classified-name` *method name* **&optional** *error-if-named* [Function]

If *method* has its stage and place notation set, sets its `method-name` to be *name* suitably augmented with its class and other modifiers, mostly from the CCCBR taxonomy (see [method-classification], page 38). That is, *name* will be what the CCCBR calls its "name", and its `method-name` will be set so that its `method-title` will usually be as the CCCBR dictates. *name* may not be null, but may be an empty string. If *method*'s stage or place notation is not set `set-method-classified-name` does nothing. Returns *method*.

Both because of ambiguities in the CCCBR Decsisions, for example around complex combinations of hunt bell types, and that Roan supports jump changes, there are some methods that may not be classified exactly as the CCCBR would dictate. Since the CCCBR doesn't recognize ringing incorporating jump changes there are no standards for the name of methods containing them, and there have been a variety of styles of names applied by those bands that do enjoy them: `set-method-classified-name` always just appends "jump" to *name*. methods containing jump changes

If the generalized boolean *error-if-named* is true, the default, an error will be signaled if *method* already has its name set and that name is not `string-equal` to the one `set-method-classified-name` will set it to. If *error-if-named* is false any existing name will be superseded.

Signals a `type-error` if *method* is not a `method` or if *name* is not a string. Signals a `parse-error` if *method*'s place notation cannot be parsed at *method*'s stage. Signals an error if *error-if-named* is true, *method* already has a name and `set-method-classified-name` would change that name.

```
(let ((meth (method :stage 8 :place-notation "x34x45x34,2")))
  (set-method-classified-name meth "Trafalgar")
  (method-name meth))
    ⇒ "Trafalgar Differential Little Alliance"
(method-title
  (set-method-classified-name
    (method :stage 12 :place-notation "x1x4,2")
    ""))
       ⇒ "Little Bob Maximus"
(method-title
  (set-method-classified-name
    (method :stage 9 :place-notation "147.(13)(46)(79)")
    "Roller Coaster"))
       ⇒ "Roller Coaster Jump Caters"
```

`cccbr-name` *method-or-string* [Function]

Strips the class name and Little and Differential modifiers off a method name to leave just that portion that the CCCBR considers a method's name. The argument can be either a string or a `method`, in which latter case it's `method-name` is used. In either case the value returned is a string. It is strict about the ordering of Differential and Little, and does not strip off stage names. Signals a `type-error` if *method-or-string* is neither a method nor a string. See also [method-classification], page 38.

```
(cccbr-name "Slink Differential Little Place")
  ⇒ "Slink"
(cccbr-name "Little Bob")
  ⇒ ""
(cccbr-name "Cambridge Major")
  ⇒ "Cambridge Major"
```

`no-place-notation-error` [Type]

Signaled in circumstances when the changes constituting a method are needed but are not available because the method's place notation or stage is empty. Contains one potentially useful slot accessbile with `no-place-notation-error-method`. Note, however, that many functions that make use of a method's place notation and stage will return `nil` rather than signaling this error if either is not present.

5.1 Falseness

Most methods that have been rung and named at stages major and above have been rung at even stages, with Plain Bob lead ends and lead heads, without jump changes, and with the usual palindromic symmetry. For major, and at higher stages if the tenors are kept together, the false course heads of such methods are traditionally partitioned into named sets all of whose elements must occur together in such methods. These are traditionally called "false course head groups" (FCHs), although they are not what mathemeticians usually mean by the word "group". Further information is available from a variety of sources, including Appendix B of the CCCBR Methods Committee's XML format documentation (`http://www.methods.org.uk/method-collections/xml-zip-files/method+xml+1.0.pdf`).

Roan provides a collection of `fch-group` objects that represent these FCH groups. Each is intended to be an singleton object, and under normal circumstances new instances should not be created. They can thus be compared using `eq`, if desired. The `fch-group`s for major are distinct from those for higher stages, though their contents are closely related.

An `fch-group` can be retrieved using the `fch-group` function. The first argument to this function can be either a `row` or a string. If a `row` the `fch-group` that contains that row is returned. If a string the `fch-group` with that name is returned. In this latter case two further, optional arguments can be used to state that the group for higher stages is desired, and whether the one with just in course or just out of course false course heads is desired; for major all the `fch-group`s contain both in and out of course elements.

The `fch-group-name`, `fch-group-parity` and `fch-group-elements` functions can be used to retrieve the name, parity and elements of a `fch-group`. The `method-falseness` function calculates the false course heads of non-differential, treble dominated methods at even stages major and above, and for those with the usual palindromic symmetry and Plain Bob lead heads and lead ends, also returns the relevant `fch-group`s. The `fch-groups-string` function can be used to format a collection of `fch-group` names in a traditional, compact manner.

It is possible to extend the usual FCH groups to methods with non-Plain Bob lead heads. However, Roan currently provides no support for this.

`fch-group` [Type]

Describes a false course head group, including its name, parity if for even stages above major, and a list of the course heads it contains. The parity is `nil` for major `fch-group`s, and one of the keywords `:in-course` or `:out-of-course` for higher stages. The elements of a major `fch-group` are major `row`s while those for a higher stage `fch-group` are royal `row`s.

`fch-group` *item* **&optional** *higher-stage out-of-course* [Function]

Returns an `fch-group` described by the provided arguments. The *item* can be either a `row` or a string designator.

If *item* is a `row` the `fch-group` that contains that row among its elements is returned. If it is not at an even stage, major or above, or if it is at an even stage royal or above but with any of the bells conventially called the seven (and represented in Roan by the integer `6`) or higher out of their rounds positions, `nil` is returned. If *item* is a `row` at an even stage maximus or above, with the back bells in their home positions, it is treated as if it were the equivalent royal `row`. When *item* is a `row` neither *higher-stage* nor *out-of-course* may be supplied.

If *item* is a string designator the `fch-group` that has that name is returned. If the generalized boolean *higher-stage* is true a higher stage `fch-group` is returned and others a major one. In the case of higher stage groups if the generalized boolean *out-of-course* is true the group with the given name containing only out of course elements is returned, and otherwise the one with only in course elements. Both *higher-stage* and *out-of-course* default to `nil` if not supplied. If there is no `fch-group` with name *item* and the given properties `nil` is returned.

Signals a `type-error` if *item* is neither a `row` nor a string designator. Signals an error if *item* is a `row` and *higher-stage* or *out-of-course* is supplied.

```
(let ((g (fch-group !2436578)))
  (list (fch-group-name g)
        (fch-group-parity g)
        (stage (first (fch-group-elements g)))))
   ⇒ ("B" nil 8)
(fch-group "a1" t nil) ⇒ nil
(fch-group-elements (fch-group "a1" t t)) ⇒ (!1234657890)
```

`fch-group-name` *group* [Function]
`fch-group-parity` *group* [Function]
`fch-group-elements` *group* [Function]

Returns the name, parity or elements of the `fch-group` *group*.

The value returned by `fch-group-name` is a string of length one or two. For major groups it is always of length one, and is a letter. For higher stages if of length one it is again a letter, and if of length two it is a letter followed by the digit '`1`' or the digit '`2`'. The case of letters in `fch-group` names is significant.

For major `fch-group`s `fch-group-parity` always returns `nil`. For higher stage `fch-group`s it always returns either `:in-course` or `:out-of-course`.

The `fch-group-elements` function returns a list of `row`s, the elements of the group. For major groups these are always major `row`s, and for higher stage groups royal `row`s. The `alter-stage` fucntion (see [alter-stage], page 14) can be helpful for making such `row`s conform to the needs of other stages above major.

All three functions signal a `type-error` if *group* is not a `fch-group`.

`fch-groups-string` *collection* **&rest** *more-collections* [Function]

Returns a string succinctly describing a set of `fch-group`s, in a conventional order. The set of `fch-group`s is the union of all those contained in the arguments, each of which should be a sequence or `hash-set`, all of whose elements are `fch-group`s. The resulting string contains the names of the distinct `fch-group`s. If there are no groups `nil`, rather than an empty string, is returned.

For higher stages there are two sequences of group names in the string, separated by a solidus ('`/`'); those before the solidus are in course and those after it out of course. For example, `"B/Da1"` represents the higher course in course elements of group B and out of course elements of groups D and a1.

The group names are presented in the conventional order. For major the groups containing in course, tenors together elements appear first, in alphabetical order; followed by those all of whose tenors together elements are out of course, in alphabetical order; finally followed by those all of whose elements are tenors parted. For higher stages the capital letter groups in each half of the string come first, in alphabetical order, followed by those with lower case names. Note that a lower case name can never appear before the solidus.

Signals a `type-error` if any of the arguments are not sequences or `hash-set`s, or if any of their elements is not an `fch-group`. Signals a `mixed-stage-fch-groups-error` if some of the elements are major and some are higher stage `fch-group`s.

```
(fch-groups-string (list (fch-group "a") (fch-group "B")))
  ⇒ "Ba"
(fch-groups-string #((fch-group "D" t t)
                     (fch-group "a1" t t))
                   (hash-set (fch-group "B" t)))
  ⇒ "B/Da1"
(fch-groups-string (list (fch-group "T" t nil)))
  ⇒ "T/"
(fch-groups-string (list (fch-group "T" t t)))
  ⇒ "/T"
```

`method-falseness` *method* [Function]

Computes the most commonly considered kinds of internal falseness of the most common methods: those at even stages major or higher with a single hunt bell, the treble, and all the working bells forming one cycle, that is, not differential. Falseness is only considered with the treble fixed, as whole leads, and, for stages royal and above, with the seventh (that is, the bell roan denotes by `6`) and above fixed. Returns three values: a summary of the courses that are false; for methods that have Plain Bob lead ends and lead heads and the usual palindromic symmetry, the false course head groups that are present; and a description of the incidence of falseness.

The first value is a list of course heads, `row`s that have the treble and tenors fixed, such that the plain course is false against the courses starting with any of these course heads. Rounds is included only if the falseness occurs between rows at two different positions within the plain course. Course heads for major have just the tenor (that is, the bell represented in Roan by the integer `7`) fixed, while course heads for higher stages have all of the seventh and above (that is, bells represented in Roan by the integers `6` and larger) fixed in their rounds positions.

If *method* has Plain Bob lead ends and lead heads, and the usual palindromic symmetry, the second value returned is a list of `fch-group` objects, and otherwise the second value is `nil`. Note also that for methods that are completely clean in the context used by this function, for example plain royal methods, an empty list also will be returned. These two cases can be disambiguated by examining the first value returned.

There is some ambiguity in the interpretation of "A" falseness. In Roan a method is only said to have "A" falseness if its plain course is false. That is, the trivial falseness implied by a course being false against itself and against its reverse by virtue of containing exactly the same rows is not reported as "A" falseness. "A" falseness is only reported if there is some further, not-trivial falseness between rows at two different positions within the plain course.

The third value returned is a two dimensional, square array, each of the elements of that array being a possibly empty list of course heads. For element e, the list at m,n of this array, lead m of the plain course of *method* is false against lead n of each of the courses starting with an element of e. The leads are counted starting with zero. That is, if s is the stage of *method*, then $0 \leq m < s\text{-}1$ and $0 \leq n < s\text{-}1$.

A `type-error` is signaled if *method* is not a `method`. Signals a `parse-error` if the place notation string cannot be properly parsed as place notation at *method*'s stage.

If *method* does not have its stage or place-notation set a `no-place-notation-error`. If *method* is not at an even stage major or above, does not have one hunt bell, the treble, or is differential, an `inappropriate-method-error` is signaled.

```
(multiple-value-bind (ignore1 groups ignore2)
    (method-falseness
      (method :stage 8
              :place-notation "x34x4.5x5.36x34x3.2x6.3,8"))
  (declare (ignore ignore1 ignore2)
  (fch-groups-string groups))
    ⇒ "BDacZ"
(fch-groups-string
  (second
    (multiple-value-list
      (method-falseness
        (lookup-method "Zorin Surprise" 10)))))
   ⇒ "T/BDa1c"
```

5.2 Methods database

Roan provides an SQLite database (`https://www.sqlite.org/`) of method definitions, containing the same methods as in the ringing.org web site (`http://www.ringing.org/methods/`). Because use of SQLite requires installation of a binary library, `libsqlite3` available from the SQLite web site (`https://www.sqlite.org/`), some Roan users may perfer to avoid it. Roan, without any of the facilities in this section for looking up methods, can be loaded by using the system `roan-base` instead of the full `roan`. Everything else in Roan should work fine without the lookup functions, and any calls to them will result in an error indicating that they are not loaded.

In addition to all the methods recognized by the CCCBR at the time Roan's database was last updated, the database also contains a variety of methods not CCCBR recognized, or, in some cases, with names or classifications differing from those with the CCCBR's imprimatur. For example, it contains methods with jump changes; methods that the Council does not consider distinct from others (for example, New Grandsire); sometimes attaches commonly-used names to methods in addition to those blessed by the Council (for example, what the Council calls Bastow Little Bob Doubles is in the database under that name, and also under St Helen's Doubles and Cloister Doubles); and methods that were disavowed by the Council because they were rung and named in a peal that did not meet the Council's requirements at the time is was rung (for example, Brindle Bob Royal, which itself met the Council's requirements at the time it was rung in spliced, but the peal contained another method, then illegal but now perfectly acceptable even to the Council). While this database does extend what the Central Council's collection of methods provides, it is still not as complete and comprehensive as it could be, though I hope it is at least helpful.

This database can be interrogated with the `lookup-methods-by-name`, `lookup-method`, `lookup-methods-by-notation` and `lookup-methods-from-changes` functions.

The methods in the database provided by Roan are a collation of methods extracted from several different sources

- The Central Council of Church Bell Ringers Method's Committee's collection of methods. This collection is copyright by the Central Council of Church Bell Ringers, and the method definitions extracted from it have been modified: their place notation and other details have been reformated, and they have been merged with method definitions from other sources.
- Tony Smith's Collection of Provisionally Named Methods. This collection is copyright by Tony Smith, and the method definitions extracted from it have been modified: their place notation and other details have been reformated, and they have been merged with method definitions from other sources.
- The Central Council's 1988 *Collection of Plain Methods*,
- Steven Coleman's *The Method Ringer's Companion*, 1995,
- Peter Hinton's Palindromic Plain Doubles Methods web page,
- a variety of helpful messages on the ringing-theory mailing list,
- and tradition and word of mouth.

`lookup-methods-by-name` *name* **&key** *stage limit update url database busy-timeout* [Function]

`lookup-method` *name* **&optional** *stage* [Function]

`lookup-methods-by-notation` *place-notation* **&key** *stage rotations limit update url database busy-timeout* [Function]

`lookup-methods-from-changes` *changes* **&key** *rotations limit update url database busy-timeout* [Function]

The `lookup-methods-by-name` function returns a list of `method`s of a given *stage* and with a name matching *name*, or an empty list if there are no such methods in the database. If the *stage* argument is not provided it defaults to the current value of `*default-stage*`. Comparison is done case-insensitively. Note that, just like for `method` objects, the name in this function is not the same as the name in the CCCBR's taxonomy: it includes what the CCCBR calls the class as well as appropriate modifiers such as "Differential" and "Little". That is, for purposes of this function, the name is the method's entire title with the stage removed. So, for example, the name of Cambridge Surprise Major is "Cambridge Surprise", and that of Baldrick Differential Little Bob Cinques is "Baldrick Differential Little Bob".

The *name* argument to `lookup-methods-by-name` can include wildcard characters: '`?`' matches any one character, and '`*`' matches any sequence of zero or more characters. To include a '`?`' or '`*`' in a string, not as a wildcard, escape it with a backslash '`\`'. Note that in Lisp literal strings you must backslash escape a backslash so you will typically end up with two backslashes. To include a backslash in a *name* pattern escape it with a backslash; in a Lisp literal string this will be a total of four backslashes. Any other use of backslash in a *name* pattern other than escaping '`?`', '`*`' or '`\`' signals an error.

For the common case of looking up a single method by name the function `lookup-method` is available. If a `method` is found it is returned and otherwise `nil` is returned. Apart from the *name*, which may not contain wildards, and the optiional *stage*, which defaults to the current value of `*default-stage*`, no other arguments may be supplied. Apart from not allowing wildcards, `lookup-method`

behaves similarly to `lookup-methods-by-name` when *limit*, *update*, *url*, *database* and *busy-timeout* are all `nil` or unsupplied.

The `lookup-methods-by-notation` function returns a list of `method`s of a given *stage* and with a plain lead defined by the place notation *notation*, a string, or an empty list if there are no such methods in the database. If *stage* is not supplied it defaults to the current value of `*default-stage*`. If the generalized boolean *rotations* is true it also returns any methods whose changes are a rotation of those given. While the CCCBR recognizes only one method with any given notation, or rotation thereof, Roan's database may contain multiple such. Wildcards cannot be used in *notation*.

The `lookup-methods-from-changes` function returns a list of `method`s with a given list of *changes* constituting its plain lead. All the elements of the list *changes* must be `row`s, and be of the same stage. The *rotations* argument is as for `lookup-methods-by-notation`.

There is no guarantee of what order methods are in the lists returned by any of these three functions.

For any of these functions the *limit* keyword argument can be used to limit the number of methods returned. If supplied it should be a positive integer. If `nil`, the default, there is no limit.

For any of these functions the *update* argument can be used to ask that the database be updated from ringing.org before querying it. Possible values for *update* are:

`nil` (the default)
: no updating is done: whatever version of the database is already present is used as is

`:query` or `t`
: ringing.org (`http://www.ringing.org/methods/`) is queried before executing the lookup, and, if it appears the version of the database on the server is more recent than that already on the local machine, a fresh copy is downloaded from the web site and used to replace the version already present

a non-negative integer
: a time duration, in units of seconds: only if at least this much time has elapsed since the version on the local machine was created is the server queried and, if a new version is present there, downloaded; the time the database on the server was created is not the same as the created or modified timestamp on the local file, nor the time it was download to the local machine

`:force`
: the most recent version on the server is downloaded, regardless of whether or not it appears to be any more recent than the version already on the local machine

An error is signaled if *update* has any other value. If the database does not exist on the local machine an `error` is signaled if *update* is `nil`, and otherwise an attempt is made to download and install it.

Unfortunately the libraries required to download and unzip an updated libary do not currently (as of June 2017) work in CLISP or LispWorks. An attempt to update

the the database with a non-nil value of *update* will signal an error in these Lisp implementations. See [update-methods-database], page 48.

The *url* argument can be used to specify a different location for downloading a fresh database, if *update* indicates that such an action should be taken. If *update* is `nil`, or otherwise indicates that the server need not be queried, *url* is ignored. If `nil` the default location is used, and otherwise *url* should be a string.

The *database* argument enables use of a different database than Roan's default. It should be a pathname designator for an existing file which is an SQLite database with the same schema as Roan's default database. See [with-methods-database], page 49, for further information on such databases. If *database* is `nil`, the default, Roan's default location for the database is used.

The *busy-timout* argument specifies how long to wait, in milliseconds, for a locked database; if `nil`, the default, operations on a locked database fail immediately.

There is no guarantee that any `method` object returned by any of these functions is distinct from that returned by a different call to the same one or a different one that needs to return such an object describing the same underlying method. For example, if two different invocations of one or two of these methods are asked to provide a definition for Cambridge Surprise Majoy the resulting `method` objects may or may not be `eq`, and subsequent changes to one may or may not be reflected in the "other" (since it may or may not be the same one).

A `type-error` is signaled if *stage* is not a `stage`; *name* or *notation* is not a string; *changes* is not a non-empty list of `row`s; *limit* is neither `nil` nor a positive integer; *update* is not of any of the types itemized above; *url* is neither `nil` nor a string; *database* is not a pathname designator; or *busy-timeout* is neither `nil` nor a non-negative integer. A `parse-error` is signaled if *notation* is a string and is not parseable as place notation at *stage*. An `error` is signaled if *name* contains a '\' followed by anything other than '`?`', '`*`' or '\'; or if *changes* is a list of `row`s, but they are not all of the same stage.

A variety of SQLite, file system or network errors may be signaled if there is difficulty opening the database file or, if necessary, reaching the server to download a fresh database.

```
(method-place-notation
  (lookup-method "Advent Surprise" 8))
    ⇒ "36x56.4.5x5.6x4x5x4x7,8"
(method-place-notation
  (first (lookup-methods-by-name "A?ve?t Sur*e" :stage 8)))
    ⇒ "36x56.4.5x5.6x4x5x4x7,8"
(method-property
  (lookup-methods-by-name 8 "Advent Surprise" :stage 8))
  :first-tower)
    ⇒ "1988-07-31 at Boston, Massachusetts, USA (Advent)"
(lookup-methods-by-name "No-such-method-anywhere" 8) ⇒ nil
(lookup-method "No-such-method-anywhere" 9) ⇒ nil
(length (lookup-methods-by-name 8 "Advent *")) ⇒ 3
(length (lookup-methods-by-name 8 "Advent *" :limit 2))
  ⇒ 2
```

```
(method-title
  (first
    (lookup-methods-by-notation 8 "36x56.4.5x5.6x4x5x4x7,8")))
      ⇒ "Advent Surprise Major"
(method-title
 (first (lookup-methods-by-changes '(!214365 !132546))))
   ⇒ "Original Minor"
(mapcar #'method-title
 (lookup-methods-by-notation 5 "5,1.3.1"))
   ⇒ ("Bastow Little Bob Doubles"
              "Cloister Doubles"
              "St Helen's Doubles")
(mapcar #'method-title
       (lookup-methods-by-notation "3,1.5.1.5.1"
                                   :stage 5
                                   :rotations nil))
  ⇒ ("Grandsire Doubles")
(mapcar #'method-title
       (lookup-methods-by-notation "3,1.5.1.5.1"
                                   :stage 5
                                   :rotations t))
  ⇒ ("Grandsire Doubles" "New Grandsire Doubles")
```

`update-methods-database` **&key** *since force url database* [Function]

Possibly downloads a fresh copy of the methods database from the ringing.org server (`http://www.ringing.org/methods/`).

If *since* is not `nil` it should be a non-negative integer. If fewer than that number of seconds have passed since the database was created, as recorded in the database, `update_methods-databse` will return `nil` immediately, doing nothing further.

Typically, before downloading the database, `update-methods-database` queries the server to see if the copy thereon differs from the local copy, and does not download it if they are the same. If *force* is not `nil`, however, it will always attempt to download a fresh copy of the database, even if it appears to be unchanged.

However, if the database does not exist on the local machine both *since* and *force* are ignored, and in this case an attempt is always made to download the database. That is, if the database does not exist locally the behavior is as if *since* were `nil` and *force* were `t`.

For example,

```
(update-methods-database :since (* 24 60 60 30))
```

will download a fresh database only if the current one was created at least thirty days ago, and the version on the server is different than the one currently stored locally; or if there is currently no such database on the local machine.

If *url* is `nil`, the default, a standard location from which to download the database is used. This can be changed by providing a URL as a string as *url*.

If *database* is `nil`, the default, a standard location is used for the database. This can be overriden by supplying a pathname designator as *database*.

Returns the pathname of the database if a fresh database is downloaded and `nil` otherwise.

Signals a `type-error` if since is neither `nil` nor a non-negative integer; *url* is neither `nil` nor a string; or *database* is neither `nil` nor a pathname designator.

Unforunately some of the libraries (Drakma, usocket and zip) required to download and unzip the updated database either don't install or no longer work in CLISP and LispWorks, as of June 2017. On these implementations a call to `update-methods-database` will result in an error. Several work arounds are practical:

- The version of the database that is downloaded with Roan will have been reasonably current at the time that version of Roan was released. For many purposes there may be no need to update it.
- Roan can be used to update it in a different Lisp implementation, such as SBCL or Clozure CL: the copy of the database so updated will then be available to CLISP or LispWorks.
- The database can be downloaded by hand. Consult the source code for `update-methods-database` for the URL, and where to put the file after unzipping it. If simply unzipped and not altered it will work for all purposes except searching for rotations of exisiting methods. To support that as well another column of one of the tables in the database must be populated after downloading the new database; in addition some indecies are created to speed up access slightly. This, too, can easily be done by hand: again, consult the source code.

`methods-database-attributes` **&optional** *database* [Function]

Returns five values describing the Roan methods database:

- the truename of the database file
- the date and time the database was created, as a univeral time; note that this will typically not be be same as the time the file was created nor the time it was downloaded
- a string (the ETag) representing that state of the file on the server, typically used to determine if an updated version is available
- the number of methods (that is, distinct stage and name pairs) that the database contains
- a vector, indexed by stage, of the number methods at each stage that the database contains

If *database* does not name a database currently on the local machine a single value, `nil`, is returned. If *database* is `nil` or not supplied the default location for Roan's database is used.

Signals a `type-error` if *database* is neither `nil` nor a pathname designator. May also signal a variety of file system or SQLite errors if the database cannot be opened or is not in the correct format.

`with-methods-database` (*var* **&key** *database busy-timeout*) **&body** *body* [Macro]

For more complex uses of Roan's methods database than the various `lookup-methods-` facilitate SQL queries can be made against the database directly

using the CL-SQLITE API (`https://common-lisp.net/project/cl-sqlite/`). The `with-methods-database` macro binds *var* to a database handle to Roan's methods database and executes *body*, returning the result; returns `nil` if *body* is empty. The database handle is closed on exit.

If *database* is supplied it should be a pathname designator to a database with the Schema of Roan's methods database. If `nil`, the default, the standard location for the database is used.

If *busy-timeout* is supplied it should be a non-negative integer, the number of milliseconds to wait when attempting to operate on a busy database. If `nil`, the default, operations on a busy database fail immediately.

Signals a `type-error` if *database* is neither `nil` nor a pathname designator, or if *busy-timeout* is neither `nil` nor a non-negative integer. A variety of SQLite or file system errors may be signaled if there is difficulty opening the database file.

```
(with-methods-database (conn)
  (iter (for (stage) :in-sqlite-query
             "select stage from methods
               where name = 'Cambridge Surprise'"
            :on-database conn)
        (collect stage)))
  ⇒ (6 8 10 12 14 16)
```

The Roan methods database has the following schema:

```
CREATE TABLE methods (stage int not null,
                      name text not null,
                      notation text not null,
                      canonicalRotation text,
                      firstTower text,
                      firstHand text);
CREATE TABLE version (updated text non null, etag text);
CREATE UNIQUE INDEX key on methods (stage, name);
CREATE INDEX notationIndex on methods (notation);
CREATE INDEX rotationIndex on methods (canonicalRotation);
```

The actual methods data is stored in the `methods` table. The `version` table should contain only a single row, and records information used by `update-methods-database` to determine whether or not the database needs updating. Neither the indices nor the contents of the canonicalRotation column exist in the copy of the database on the server, and are instead created by `update-methods-database` after it is downloaded to reduce the size of the file transferred over the network.

The `stage` and `name` columns of the `methods` table are the stage and name of the method described by that row of the table (polysemy alert: a table row is unrelated to a change ringing row), and together form a primary key for that table; that is, there can be no two rows that have the same `stage` and `name`.

The `notation` column is the corresponding place notation, in a canonical form with internal places elided as by `:elide :lead-end` in `write-place-notation`, capital letters used for place eleven or above, a lower case '`x`' used for cross, and palindromic methods with an even lead length unfolded as by `:comma t`.

The `canonicalRotation` is the place notation corresponding to a unique rotation of of changes of the method. That is, if two methods have the same changes, up to rotation, they will have the same `canonicalRotation`. Note that this rotation is in no way a preferred one, and is in most cases one few bands would ever care to ring; it is just a way of comparing two methods to see if they are the same up to rotation. The place notation for `canonicalRotation` is in the same canonical form as `notation`.

The `firstTower` and `firstHand` columns are succinct, textual representations of when, and possibly where, the first peal in the methods occurred in tower or hand, respectively, if that information is available in the database. Sometimes it is not available, or no such peal has occured, in which the column is `null`. See [canonicalize-method-place-notation], page 31, See [write-place-notation], page 16,

Appendix A License

Roan is covered by an MIT open source license (`https://en.wikipedia.org/wiki/MIT_License`), a well-known, permissive license with few compatibility problems with other licenses. The license is quoted below. Loading Roan also loads several third party libraries, each of which is made available under its own terms, distinct from Roan's. To the best of my understanding all the libraries Roan loads are either in the public domain, or have similarly permissive licenses; however, you should read their actual licenses to be sure. See [dependencies], page 53.

Roan's license is:

Appendix B Libraries Used by Roan

Roan loads the following libraries. Note that it does not include them as part of itself, it merely loads them, typically over the network from Quicklisp's library server.

While most are in the public doamin or offered under a permissive, open source license, two are offered with a copyleft license: ZIP and S-BASE64. This should make no difference for using Roan in the usual way, with the libraries loaded from Quicklisp. But if you distribute something built with Roan and include the libraries in it you will have to pay attention to these licenses and be sure to adhere to them.

- Alexandria (`https://common-lisp.net/project/alexandria/draft/alexandria.html`) [public domain]
- CL-PPCRE (`http://weitz.de/cl-ppcre`) [BSD]
- Drakma (`http://weitz.de/drakma/`) [BSD]
- CL-FAD (`http://weitz.de/cl-fad/`) [BSD]
- Iterate (`https://common-lisp.net/project/iterate`) [MIT]
- CL-SQLITE (`https://common-lisp.net/project/cl-sqlite/`) [public domain]
- Trivial Garbage (`https://common-lisp.net/project/trivial-garbage`) [public domain]
- ZIP (`https://common-lisp.net/project/zip/`) [BSD+LGPL]
- S-BASE64 (`https://github.com/svenvc/s-base64`) [LLGPL]

Several further libraries are used in the construction of Roan, but their license terms should not affect those who use or ship Roan itself, except possibly in so far as they need to use them to develop modifications to Roan.

While not loaded during normal use of Roan, when running Roan's unit tests (see [testing], page 56) the following library is also loaded and used.

- lisp-unit2 (`https://github.com/AccelerationNet/lisp-unit2`) [MIT]

Also while not loaded during normal use of Roan, when building Roan's documentation (see [building], page 55) the following libraries are also loaded and used. One of these, too, `trivial-documentation`, is offered under a copyleft license. To the best of my understanding this should not affect those simply copying or distributing Roan, so long as they do not include a copy of `trivial-documentation` with it.

- CL-FAD (`http://weitz.de/cl-fad/`) [BSD]
- trivial-documentation (`https://github.com/eugeneia/trivial-documentation`) [Affero GPL]

While they are not loaded into Lisp, a variety of other tools are used in building Roan's documentation, some of which are distributed with copyleft licenses. My understanding is that this shouldn't be an issue so long as you don't redistribute or modify them, however.

Appendix C History

Roan started out in the mid-1970s as code to support searching for compositions using Portable Standard Lisp on a Digital Equipment Corporation DEC-10 machine. A few years later a bunch of utilities for UNIX, written in C, were added. By the mid-1980s it had expanded significantly into collections of Apollo Domain Lisp and Lisp Machine Lisp code. From there to a highly portability-challenged version for Digitool Macintosh Common Lisp, followed by a more portable version that I used for several years, predominately with CLISP. Over the years I've even "ported" it (if you can call a complete rewrite in a different language a "port") to a few other programming languages where I used it for various lengths of time. Eventually I tried to make a more tidy, fairly portable Common Lisp version, and have been happily using this for my compositional activities in recent years. It also underpins most of the method presentation stuff on the ringing.org (`http://www.ringing.org/`) web site.

Over the years several folks received copies of it, and in some cases used it, at least for a little while. I think it was used for a time on a Lisp Machine to drive a set of electronic Christmas tree ornamental bells ringing touches! But despite such use, it was never well organized or documented.

In a fit of good-neighorliness I've finally tried to make it sufficiently tidy for more general distribution and added this documentation, in the hopes that others may find it useful, too.

As you can easily deduce from its history, lots of things have come and gone over the years, and there are lots of parts that have fallen into disrepair or don't play well with other parts. In tidying it up for distribution I'm trying to fix that. Public releases of Roan will contain only portions that have been reasonably well tested and that do play well together, but there's still a lot of work to do resurrecting, tidying and documenting other features of varied antiquity and robustness. I'm hopeful that in coming months and years I'll be able to offer more releases with more good stuff added to them. Two additions that should be coming soon are a fairly general notion of calls amending the changes of leads of methods, and a mechanism for drawing bluelines.

C.1 What's with the name?

Robert Roan was an important seventeenth century ringer and composer. He is believed to have invented Grandsire Doubles and Plain Bob Minor, and by implication, the "standard extent" of minor.

Sadly, his name is less well known among most ringers than some other early composers, probably because he's had the misfortune of never having had a method named after him. So several decades ago it seemed fitting to name a library of software intended for use in composition after him. In keeping with Robert Roan's relative obscurity, it's taken several decades for the eponymous software to become publicly available.

Appendix D Building and Modifying Roan

Since Roan is written in Lisp there really is no build process: your Lisp implementation, under the direction of Quicklisp and ASDF, will happily just compile and load it.

However, the documentation does have a slightly complex build process. And if you are modifying Roan you really should make friends with the unit tests.

D.1 Building the documentation

This manual is written using Texinfo (`https://www.gnu.org/software/texinfo/`). Depending upon how your environment is already configured you may have to download and install Texinfo software, TeX and/or LaTeX.

There is a file included in the source tree, though not a part of Roan per se, `extract-documentation.lisp`, that is used to extract the documentation strings associated with symbols exported from the `roan` package. This is done using the `roan/doc:extract-documentation` function. For each exported symbol it writes a small `.texi` file in the `doc/inc` directory with its documentation string, augmented with some further Texinfo commands. The documentation strings in the Roan sources are themselves infested with appropriate Texinfo commands. Most of these small files are then `@included` into the main Texinfo file, `roan.texi`, to document the various functions, macros and types. In addition to `roan.texi`, there is also a small collection of style information used by the HTML versions of the documentation in `roan.css`.

After the documentation strings have been extracted `makeinfo` needs to be called for each of the four versions of the documentation that are produced:

- an Info file
- a PDF file
- a single HTML file
- a collection of HTML files, one per chapter

So long as CCL (Clozure Common Lisp) is installed, the `Makefile` in the source hierarchy should do all this for you. If preferred, it should be straightforward to modify the `Makefile` to use a different Lisp implementation to run `extract-documentation`.

If a public function is added to Roan, it should include a suitable documentation string, and `roan.texi` should be revised to give it an appropriate home, where the corresponding `inc/*-function.texi` file is `@included`. So long as you export its name from the `roan` package it should be picked up by when building the documentation. Then, run `make documentation` and if all goes well[1], all four kinds of documentation will be nicely produced a few seconds later.

Of course, most of the time, all won't go well. Besides looking carefully at the rather noisy output from `makeinfo`, here are a few other things to bear in mind:

- Even though a problem looks like it's in `roan.texi`, it might be in a documentation string in a source file; Texinfo is pretty good about identifying the include file that's causing the problem, unless the problem is just too subtle for it.

[1] "If all goes well" is reputed to have been a favorite expression of the sea captain Edward Smith.

- Any occurrences of braces, '`{}`', or an at-sign, '`@`', in a documentation string will be interpreted magically by Texinfo, and need to be quoted with an '`@`'.
- When writing examples in documentation strings, if the examples use Lisp strings, the double quotes need to be escaped with back slashes, or Lisp will become confused.
- Even if you are not one yourself, be nice to Emacs users. In Lisp mode Emacs treats an open parenthesis, '`(`', at the beginning of a line specially, and will become badly confused if you have such in a documentation string. When writing examples, indent such Lisp code by one space to keep Emacs happy. Similarly be sure to format documentation strings so that parenthetical comments do not start at the beginning of a line; Emacs's own `fill-paragraph` command is careful about that on your behalf.
- Texinfo is a complex system. If you're going to use it, you're going to have to learn a bit about it. Until you become facile with it its usually easiest to make changes slowly and incrementally.
- Writing good documentation is often harder than writing the corresponding code.

D.2 Running unit tests

The Roan source tree contains a collection unit tests. If you are making changes to Roan it is *highly* recommended that you make friends with them and use them early and often. Unless run on an unusally slow machine or Lisp implementation it takes only a few seconds to run the full collection, and in the long run they will save you a lot of time.

The unit tests live in their own package, `roan/test`, and make use of a further library not included in Roan itself, lisp-unit2 (`https://github.com/AccelerationNet/lisp-unit2`). After running `(ql:quickload :roan)` once, run `(ql:quickload :roan/test)` once as well. Once that has been done, assuming Roan has been installed where ASDF can find it, it should be possible to run all the unit tests by simply evaluating `(asdf:test-system :roan)`. If the unit tests all succeed it will report the success of about 13,000 assertions with no failures. Otherwise, you'll have some debugging to do.

A few caveats:

- The unit tests assume the full version of Roan, `(ql:quickload :roan)`, is installed, not just `:roan-base`.
- A few of the tests, related to upgrading the methods database, require access to the internet. If it is not available, there will be a failure reported, and a few tests will be skipped.
- The tests related to upgrading the methods database depend upon a couple of test databases hosted on ringing.org. While these databases are not expected to change frequently, it is possible that they may have to change at some point, making them no longer compatible with older versions of Roan; in such a case for all the unit tests to pass you will have to be using a recent enough version of Roan.

If you make changes to Roan I suggest you run the unit tests frequently. If you care about portability, run them at least occasionally on as many different Lisp implementations as you have access to. And if you add functions to Roan, add unit tests for them, too.

Writing decent, reusable tests is often harder and more time consuming than writing code; but it's a lot easier and less time consuming than debugging things days, weeks,

months or years after they were first written. This is especially true for a relatively low-level library such as Roan.

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

X