Roan

Version 9.0.19

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 library of method definitions, together with a mechanism for updating that library from the CCCBR Methods Library (`https://cccbr.github.io/methods-library/`)
- a function for drawing blue lines of methods as Scalable Vector Graphics (SVG) images
- a function for extracting false course heads from common kinds of methods
- a representation of calls, allowing replacing, deleting or adding one or more changes to a plain lead of a method, at any point within that lead, and possibly spanning two leads as is done in doubles variations.

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 55. Roan also loads and uses a variety of other libraries. See [dependencies], page 56.

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`.

Note that Quicklisp creates a new distribution about once a month, so there may be a log of that duration between when a new version is available in the Bitbucket repository and when that version is available in Quicklisp. If you need it sooner, you may need to download it yourself from Bitbucket.

Roan has been tested with at least

- CCL (Clozure CL) (`http:ccl.clozure.com`) version 1.11.5 (64 bit), on Ubuntu Linux 18.04.3,
- and SBCL (Steel Bank Common Lisp) (`http://sbcl.org`) version 1.5.5 (64 bit) on Ubuntu Linux 18.04.3.

but should also work in other, modern Common Lisp implementations that support the libraries on which Roan depends. See [dependencies], page 56.

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 to 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))`
- if the problem might involve the lookup of methods, then also the output from `(roan:method-library-details)`
- 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 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`, `method-name`, `class` and `class-name` from the `common-lisp` package. This is done because methods and their classes are important concepts in change ringing, albeit one unrelated to CLOS methods and classes. Typically `method`, `method-name`, `class` and `class-name` should be shadowed in other packages that use the `roan` package. This can be done with `shadowing-import-from`, or the `:shadowing-import` option to `defpackage`. Note that the original Common Lisp symbols will still be available as `cl:method`, `cl:method-name`, `cl:class` and `cl:class-name`. See [use-roan], page 4.

```
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 three symbols from the `common-lisp` package: `method`, `method-name`, `class` and `class-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` **&key** *package syntax modify* [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, shadowing `method`, `method-name` and `class-name`.

If the generalized boolean *syntax* is true, the default, it also enables use of Roan's '`!`' and '`#!`' read macros, by calling [roan-syntax], page 7, with a true first argument; the value of *modify* is passed as a second argument to [roan-syntax], page 7.

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> (rowp '!13276548)
NIL
MY-PACKAGE> (roan:use-roan)
T
MY-PACKAGE> +maximum-stage+
24
MY-PACKAGE> (rowp '!13276548)
T
```

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 8, [`row-string`], page 8, [`write-place-notation`], page 17, and [`place-notation-string`], page 18.

2.3 Rows

The fundamental units of change 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 can be 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 rows are Lisp atoms, and thus literal values can be written using '!' notation without quoting, though quoting rows read that way will do no harm when they are evaluated.

This '!' syntax can be turned on and off by using [roan-syntax], page 7. By default it is off when Roan is loaded. It is also possible to control this syntax by using Named Readtables (https://github.com/melisgl/named-readtables/); see [roan-syntax], page 7, for further details.

Similarly, rows 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 bells 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 rows 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 rows. See [hash-set], page 20.

```
(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.

roan-syntax **&optional** *on-off modify* [Macro]

Turns on or off the read macros for '!' and '#!', for reading rows and place notation.

If the generalized boolean *on-off* is true, the default, it turns on these read macros. Unless the generalized boolean *modify* is false, the default, it first pushes the current read table onto a stack, modifying a copy of it and making that copy the current read table. If *modify* is true it makes no copy and instead modifies the current readtable in place.

If *on-off* is false it restores the previous readtable by popping the stack. If the stack is empty it sets the readtable to a new, standard one. When *on-off* is false *modify* is ignored.

This is performed in an `eval-when` context to ensure it happens at compile time as well as load and execute time.

An alternative to using `roan-syntax` is to use Named Readtables (`https://github.com/melisgl/named-readtables/`). Roan defines two such readtables with names `:roan` and `:roan+interpol`. The former augments the initial Common Lisp read table with Roan's read macros, and the latter also adds the syntax from CL-INTERPOL (`http://edicl.github.io/cl-interpol/`).

`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.

`reversed-row` *row* [Function]

Returns a `row` of the same stage as *row* with its bells in the reverse order. A `type-error` is signaled if *row* is not a `row`.

```
(reversed-row !32148765) ⇒ !56784123
```

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 differential. 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 minimus. 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 !1243) ⇒ 1
(which-plain-bob-lead-head !12345) ⇒ 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)
```

`permutation-closure` **&rest** *rows* [Function]

Returns a list of distinct rows that can be generated by permuting, repeatedly if necessary, any of the *rows* by themselves or any others of the *rows*. If the *rows* are not all of the same stage, the lower stage ones are converted to the highest stage

present before the closure operation is performed. The order of the returned rows is undefined. Signals a `type-error` if any of the *rows* is not a `row`.

```
(permutation-closure !13425 !1324 !123465)
  ⇒ (!143265 !142365 !124365 !142356 !143256 !124356
      !134265 !132465 !123456 !123465 !132456 !134256)
```

`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 can augment the Lisp reader with '`!`' to read `rows`, it can augment 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 `rows`, 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.

This '`#!`' syntax can be turned on and off by using [roan-syntax], page 7. By default it is off when Roan is loaded. It is also possible to control this syntax by using Named Readtables (`https://github.com/melisgl/named-readtables/`); see [roan-syntax], page 7, for further details.

```
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 using lower case '`x`' for cross, upper case letters for 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 `row`s of different stages. See [write-place-notation], page 17.

```
(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 `rows`. 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 `rows` 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 what the Central Council of Church Bell Ringers Framework for Method Ringing (`https://cccbr.github.io/method_ringing_framework/`) (FMR) calls a static method, a method that can be viewed as a fixed sequence of changes, including jump changes; while this includes nearly all methods rung and named to date, it does exclude, for example, Dixonoids. A `method` has a name, a stage, classifacation details, and an associated place-notation, though any or all of these may be `nil`. In the case of the stage or place notation `nil` indicates that the corresponding value is not known; the same is also true if the name is `nil`, except for the case of Little Bob, which in the taxonomy of the FMR has no name. The stage, if known, should be a `stage`, and the name and place notation, if known, should be strings.

The classification follows the taxonomy in the FMR and consists of a `class` and three boolean attributes for jump methods, differential methods and little methods. The `class` may be `nil`, for principles and pure differentials; one of the keywords `:bob`, `:place`, `:surprise`, `:delight`, `:treble-bob`, `:alliance`, `:treble-place` or`:hybrid`, naming the corresponding class; or `:hunt` indicating a method with one or more hunt bells that does not fall into any of the named classes, which can only apply to jump methods. The classification consists merely of details stored in the `method` object, and does not necessary correspond to the actual classification of the method described by the `place-notation`, if supplied. The classification can be set to match the place notation by calling `classify-method`.

Similarly the name does not necessarily correspond to the name by which the place notation is known, unless the `method` has been looked up from a suitable library. See Section 5.1 [Methods library], page 34.

Because ringing methods and their classes are unrelated to CLOS methods and classes, the `roan` package shadows the symbols `common-lisp:method`, `common-lisp:method-name`, `common-lisp:class` and `common-lisp:class-name`.

`method` [Type]

Describes a change ringing method, typically including its name, stage, classification and place notation.

`method` **&key** *name jump differential little class stage place-notation* [Function]

Creates a new `method` instance, with the specified *name*, *stage*, *classification* and *place-notation*. If *stage* is not provided, it defaults to the current value of `*default-stage*`; to create a `method` with no stage `:stage nil` must be explicitly supplied.

A `type-error` is signaled if *stage* is supplied and is neither `nil` nor a `stage`; if either of *name* or *place-notation* are supplied and are neither `nil` nor a string; or if `class` is supplied and is neither `nil` nor one of the keywords `:bob`, `:place`, `:surprise`, `:delight`, `:treble-bob`, `:alliance`, `:treble-place` or `:hybrid`. A `inconsistent-method-specification-error` is signaled if the various classification details cannot occur together, such as a little principle.

`method-name` *method* [Function]

`method-jump-p` *method* [Function]

`method-differential-p` *method* [Function]
`method-little-p` *method* [Function]
`method-class` *method* [Function]
`method-stage` *method* [Function]
`method-place-notation` *method* [Function]

Return the name, classification details, stage and place notation of *method*, or `nil`. A non-nil value returned by `method-name` or `method-place-notation` is a string; by `method-stage` a `stage` (that is, an integer); and by `method-class` one of the keywords `:bob`, `:place`, `:surprise`, `:delight`, `:treble-bob`, `:alliance`, `:treble-place`, or `:hybrid`. The predicates `method-jump-p`, `method-differential-p` and `method-little-p` return generalized booleans. These functions all signal a `type-error` if *method* is not a `method`.

Note that `method-jump-p` reflects the classification stored in *method*, while `method-contains-jump-changes-p` reflects the place notation of *method*, and they may not agree.

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. Attempting to set the name or place notation to anything but a string or `nil`, the class to anything but `nil` or one of the appropriate keywords, or the stage to anything but a `stage` or `nil` signals a `type-error`. See [method-contains-jump-changes-p], page 39,

`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. Signals a `type-error` if *method* is not a `method`.

The one argument case can be used with `setf`, in which case it potentially sets any or all of the name, classification and stage of *method*. There is an ambiguity when parsing method titles in that there being no explicit class named can indicate with that the method has no class (principles and pure differentials) or that the class is Hybrid. When parsing titles for `setf` an absence of a class name is taken to mean that there is no class. Also, if there is no stage name specified when using `setf` with `method-title` the stage is set to `nil`; `*default-stage*` is not consulted.

```
(method-title (method "Advent" :class :surprise :stage 8))
  ⇒ "Advent Surprise Major"
(method-title (method :name "Grandsire" :class :bob :stage 9))
  ⇒ "Grandsire Caters"
(method-title (method :stage 8))
  ⇒ "Major"
(method-title (method :class :delight :stage 8) t)
  ⇒ "Unknown Delight Major
(method-title (method :name "Advent" :class :surprise :stage nil))
  ⇒ "Advent Surprise"
(method-title (method :name "Slinky" :stage 12 :class :place
                      :little t :differential t))
  ⇒ "Slinky Differential Little Place Maximimus"
(method-title (method :name "Stedman" :stage 11))
  ⇒ "Stedman Cinques"
(method-title (method :name "Meson" :class :hybrid
                      :little t :stage 12))
  ⇒ "Meson Maximus"
```

`method-from-title` *title* **&optional** *place-notation* [Function]

Creates a new `method` instance, with its name, classification and stage as specified by *title*, and with the given *place-notation*. If the title does not include a stage name, the stage of the result is the current value of `*default-stage*`.

Note that it is not possible to distinguish hybrid methods from non-jump principles, nor jump methods with hunt bells from those without, from their titles. By convention, if no hunt bell class is specified in *title* a principle, that is a method without hunt bells, is assumed. If in some specific use this is not correct it can be corrected by setting `method-class`, and possibly `method-little-p`, of the resulting method as desired.

A `type-error` is signaled if *title* is not a string, or if *place-notation* is neither a string nor `nil`.

```
(let ((m (method-from-title "Advent Surprise Major")))
  (list (method-title m) (method-class m) (method-stage m)))
  ⇒ ("Advent" :surprise 8)
```

`comparable-method-name` *string* [Function]

If *string* is a suitable name for a method, returns a version appropriate for comparison with other comparable names, and otherwise returns `nil`.

The Central Council of Church Bell Ringers Framework for Method Ringing (`https://cccbr.github.io/method_ringing_framework/`) (FMR), appendix B describes a syntax for method names and their comparisons. This function both determines whether or not they fit within the syntax described by the FMR, and, if so, provides a canonical representation for them suitable for comparing whether or not two apparently different names will be considered the same when describing a method. This comparable representation is not intended for presentation to end users, but rather just for comparing names for equivalence.

Signals a `type-error` if *string* is not a string.

```
(comparable-method-name "New Cambridge")
  ⇒ "new cambridge"
(comparable-method-name "London No.3")
  ⇒ "london no 3"
(comparable-method-name "mäkčeň E=mc²")
  ⇒ "makcen e mc2"
(comparable-method-name "Two is Too  Many Spaces")
  ⇒ nil
(comparable-method-name "Ελληνικά is Greek to me")
  ⇒ nil
```

`inconsistent-method-specification-error` [Type]

Signaled in circumstances when the various classification details provided cannot occur together, such as a little principle.

5.1 Methods library

Roan provides a library of method definitions, derived from the Central Council of Church Bell Ringers Methods Library (`https://cccbr.github.io/methods-library/index.html`). These are augmented with a handful of other methods not yet in the CCCBR Library, jump methods and common alternative names for a few methods ([lookup-method-info], page 36). As delivered with Roan this library is only up to date as of the date a version of Roan was released. However, if a network connection is available, the library can be updated to the most recent version made available by the Council by using `update-method-library`. The Council typically updates their library weekly.

The library can be interrogated with the `lookup-methods`, `lookup-method-by-title` and `lookup-methods-by-notation` functions. Additional information such as dates and places of first peals containing the methods is available for some of the methods using `lookup-method-info`.

`lookup-methods` **&key** *name jump differential little class stage* [Function]
`lookup-method-by-title` *title* [Function]
`lookup-methods-by-notation` *notation-or-changes* **&optional** *stage* [Function]

The `lookup-methods` function returns a list of named `method`s whose name, classification and/or stage match those provided. If only a subset of these properties are provided, the return list will contain all known methods that have the provided ones.

If *name* is provided, it should be a string or `nil`, and all the methods returned will have that name. The Central Council of Church Bell Ringers Framework for Method Ringing (`https://cccbr.github.io/method_ringing_framework/`) (FMR), appendix C defines the form method names may take, and a mechanism for comparing them that is more complex than simply comparing strings for equality. For example, `"London No.3"` and `"London no 3"` are considered the same names. The `lookup-methods` function uses this mechanism. See [comparable-method-name], page 33.

The *name* may also contain '`*`' wildcard characters. Such a wildcard matches a series of zero or more consecutive characters. Since the '`*`' is not a character allowed in method names by the FMR there is no ambiguity: occurrences of '`*`' in *name* are

always wildcard characters. Wildcards are applicable only to *name*, and not to any of the other arguments to `lookup-methods`.

If *stage* is provided, it should be a `stage`, that is a small integer. All the methods that are returned will have that stage. While a `method` object can have an indeterminate stage, represented by `nil`, all the methods returned by `lookup-methods` will have a definite stage, and `nil` is not an allowed value for the *stage* argument.

If *class* is provided, it should be `nil` or one of the keywords `:bob`, `:place`, `:surprise`, `:delight`, `:treble-bob`, `:treble-place`, `:alliance`, `:hybrid` or `:blank`. With the exception of `:blank`, all the methods returned will have the specified class. The value `:blank` matches either `nil`, meaning no explicit class, or `:hybrid`; when writing a method's title according to the FMR the hybrid class and no class are indistinguishable, since "hybrid" is not included in the title.

If supplied, the generalized booleans *little*, *differential* and *jump* indicate that the returned methods should or should not have these properties. If these parameters are not supplied all otherwise matching methods in the library will be returned without regard to whether or not they have these properties.

If the title of a method is known, it can be found in the library by using `lookup-method-by-title`. The *title* should be a string. If a `method` with that title is in the library, it is returned; otherwise `nil` is returned. In general there should never be two or more different methods in the library with the same title. Matching on the title is done using the FMR's mechanism for comparing names. Wildcards cannot be used with `lookup-method-by-title`.

If the place notation of a method is known, and its name in the library is sought, `lookup-methods-by-notation` is available. The *notation-or-changes* should be either a string, in which case it viewed as place notation, or a list of `rows`, representing changes all of the same stage. The *stage* should be a `stage`; if not provided or `nil` the current value of `*default-stage*` is used. If *notation-or-changes* is a list of changes, the value of *stage* is ignored, the stage of those changes being used instead. Two lists are returned. The first is of methods that have the provided place notation (or corresponding changes). The second is of methods that are rotations of methods with the given place notation. Either or both lists may be empty if no suitable methods are found in the library.

There is no guarantee of what order methods are in the lists returned by `lookup-methods` or `lookup-methods-by-notation`. Instances of the "same" method returned by different invocations of these functions will typically not be `eq`.

A `type-error` is signaled if *stage* is not a `stage` (or, in the case of `lookup-methods-by-notation`, `nil`); *name* is not a string or `nil`; *notation-or-changes* is neither a string nor a non-empty list of `rows`; *changes* is not a non-empty list of `rows`; or if *class* is not one the allowed values. A `parse-error` is signaled if *notation-or-changes* is a string and is not parseable as place notation at *stage*. An `error` is signaled if *changes* is a list of `rows`, but they are not all of stage *stage* (or of `*default-stage*` if *stage* is `nil`). A `method-library-error` is signaled if the method library file cannot be read or is of the wrong format.

```
(mapcar #'method-place-notation
        (lookup-methods :name "Advent"
                        :class :surprise
                        :stage 8))
    ⇒ ("36x56.4.5x5.6x4x5x4x7,8")
(mapcar #'method-title
        (lookup-methods :name "london no 3"
                        :class :surprise
                        :stage 10))
    ⇒ ("London No.3 Surprise Royal")
(method-place-notation
  (lookup-method-by-title "Advent Surprise Major"))
    ⇒ "36x56.4.5x5.6x4x5x4x7,8"
(lookup-methods :name "No such method")
    ⇒ nil
(mapcar #'method-title
        (lookup-methods :name "Cambridge*"
                        :class :surprise
                        :stage 8))
    ⇒ ("Cambridge Blue Surprise Major"
               "Cambridge Surprise Major"
               "Cambridgeshire Surprise Major")
(multiple-value-bind (n r)
    (lookup-methods-by-notation "36x56.4.5x5.6x4x5x4x7,8" 8)
      (list
        (mapcar #'method-title n)
        (mapcar #'method-title r)))
    ⇒ (("Advent Surprise Major") nil)
(multiple-value-bind (n r)
    (lookup-methods-by-notation "1.3" 3)
      (list
        (mapcar #'method-title n)
        (mapcar #'method-title r)))
    ⇒ (("Reverse Original Singles")
               ("Original Singles"))
(method-place-notation
  (lookup-method-by-title "Original Singles"))
    ⇒ "3.1"
```

`lookup-method-info` *title-or-method key* [Function]

Roan's method library also stores metadata about many of the methods it contains. Each kind of such metadata is described by a keyword, which is passed to this function as *key*. The *title-or-method* may be a string or a `method`. If a string, it is the title of the method about which the metadata is sought. If the metadata indicated by *key* is available for the method it is returned; the type of the return value depends upon the kind of metadata sought. If no such metadata is available, including if *key* is a

not yet supported type of metadata or if `title-or-method` does not correspond to any method in the library, `nil` is returned.

Currently supported values for *key* are

`:first-towerbell-peal`
: Returns a string describing the first performance of the method on tower bells. No distinction if made between ringing the method on its own or ringing it in spliced.

`:first-handbell-peal`
: Returns a string describing the first performance of the method on hand bells. No distinction if made between ringing the method on its own or ringing it in spliced.

`:complib-id`
: Returns an integer, which is used to index information about the method on Composition Library (`https://complib.org/`). This can also be used to distinguish those methods added to those from the Central Council, as the added methods do not have a `:complib-id`, while all those from the Council do.

Others may be added in future versions of Roan.

Signals a `type-error` if *title-or-method* is neither a string nor a `method`, or if *key* is not a keyword.

```
(lookup-method-info "Advent Surprise Major"
                    :first-towerbell-peal)
    ⇒ "1988-07-31 Boston, MA (Advent)"
(lookup-method-info
  (first (lookup-methods-by-notation "36x56.4.5x5.6x4x5x4x7,8"))
  :complib-id)
    ⇒ 20042
(lookup-method-info "Advent Surprise Major"
                    :no-such-info)
    ⇒ nil
```

`update-method-library` **&optional** *force* [Function]

Queries the remote server containing the CCCBR's Methods Library. If that remote file has changed since the one Roan's library was built from was downloaded, it fetches the new one and uses it to build an updated Roan method library. If the generalized boolean *force* is true it fetches the remote file and rebuilds Roan's library without regard to whether the remote one has changed. If the library is updated, returns an integer, the number of methods the updated library contains; if the library is not updated because the remote version hasn't changed returns `nil`.

May signal any of a variety of file system or network errors if network access is not available, or unreliable, or if there are other difficulties downloading and processing the remote file.

`method-library-details` [Function]

Returns eight values describing the current Roan method libary. All are strings. They are:

1. A description of the CCCBR Method Library, extracted from the file from which the Roan library was constructed
2. The date and time the file on the remote server was last modified, according to that server.
3. The "entity tag" (ETag) of the remote file, as provided by the server. This is an opaque identifier that changes for each version of the remote file. Querying the current Etag is how `update-method-library` decides whether or not the Roan method library needs updating.
4. The URL used to fetch the remote file from which the Roan library was built.
5. The *source-id* provided in the remote file, that is a CCCBR version stamp.
6. The date the CCCBR library was built, according to the contents of the file downloaded from the remote server. This may or may not be the same as the date the file on the remote server was last modified.
7. A unique identifier for the current version of the Roan library. This will change whenever the Roan library is rebuilt, even if the resulting contents are unchanged.
8. The date and time the current version of the Roan library was built.

`method-library-error` [Type]

Signaled when a method library file cannot be read. Contains two potentially useful slots accessible with `file-error-pathname` and `method-library-error-description`.

5.2 Functions of the place notation of methods

`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 17.

```
(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 `row`s, 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-p` *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-p` will still return `nil`.

Note that this function reflects the place notation of *method* while `method-jump-p` reflects the classification stored in the method, and they may not agree.

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-p
  (method :place-notation "x3x4x2x3x4x5,2"
          :stage 6))
    ⇒ nil
(method-contains-jump-changes-p
  (method :place-notation "x3x(24)x2x(35)x4x5,2"
          :stage 6))
    ⇒ t
(method-contains-jump-changes-p
  (method :stage 6))
    ⇒ nil
(method-contains-jump-changes-p
  (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-from-title "Little Bob Major" "x1x4,2"))
    ⇒ !16482735
```

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

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`.

Signals 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-from-title "Grandsire Doubles"
                                      "3,1.5.1.5.1"))
    ⇒ (0 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-lead-head-code` *method* [Function]

Returns the lead head code for *method*, as a keyword, if its stage and place

notation are set and it has Plain Bob or Grandsire lead ends, and otherwise returns `nil`. No methods below minimus are considered to have such lead ends, nor is rounds considered such a lead end. When not `nil` the result is a keyword whose name consists of a single letter, possibly followed by a digit.

The CCCBR's various collections of methods have, for several decades, used succinct codes, typically single letters or, more recently, single letters followed by digits, to denote various lead ends for the methods they contain. While the choices made have in the past varied by collection, in recent decades a consistent set of codes has been used, which is now codified in the Central Council of Church Bell Ringers Framework for Method Ringing (`https://cccbr.github.io/method_ringing_framework/`) (FMR), appendix C. While these codes actually describe both a row and a change adjacent to that row, and thus two different rows, the FMR calls them "lead head codes", so that phrasing is also used here.

There is currently (as of July 2019) an issue with the definitions of these codes in the FMR, where those for Grandsire-like methods do not correctly correspond to common practice. For example, most ringers would consider Itchingfield Slow Bob Doubles and Longford Bob Doubles to have the same lead ends. However, the current FMR definition says that the former has 'c' Grandsire lead ends, and the latter does not. This is currently under discussion for correction in the next revision of the FMR. The `method-lead-head-code` function is implemented assuming that this will be corrected in the next revision of the FMR to match common practice. For example,

it considers neither Itchingfield Slow Bob nor Longford Bob as having Grandsire lead ends.

It is also worth noting that, for some of the less common cases, the lead end codes defined in the FMR differ from those used in earlier CCCBR collections.

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-by-title "Advent Surprise Major"))
    ⇒ :h
(method-lead-head-code
  (lookup-method-by-title "Zanussi Surprise Maximus"))
    ⇒ :j2
(method-lead-head-code
  (lookup-method-by-title "Sgurr Surprise Royal"))
    ⇒ :d
(method-lead-head-code
  (lookup-method-by-title "Twerton Little Bob Caters"))
    ⇒ :q2
(method-lead-head-code
  (lookup-method-by-title "Grandsire Royal"))
    ⇒ :p
(method-lead-head-code
  (lookup-method-by-title "Double Glasgow Surprise Major"))
    ⇒ nil
```

`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, and only if, two methods are rotations, possibly trivially so, of one another their `method-canonical-rotation-key`s will always be `equal`. While a string, the value is essentially an opaque type and should generally not be displayed to an end user or otherwise have its structure depended upon, though it can be printed and read back in again. 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
```

`classify-method` *method* [Function]

Assigns the classification fields of *method* to match the classification assigned by the Central Council of Church Bell Ringers Framework for Method Ringing (`https://cccbr.github.io/method_ringing_framework/`) (FMR) for the place notation contained in that `method`, and returns the method. Signals a `type-error` if *method* is not a `method`. Signals a `no-place-notation-error` if either the stage or place notation of *method* are not set. Signals a `parse-error` if the value of the place notation field cannot be interpreted as place notation at the stage of *method*.

```
(method-title (classify-method
                (method :stage 8 :place-notation "x3x6x5x45,2"))
              t)
    ⇒ "Unnamed Differential Little Surprise 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.

5.3 Drawing blue lines

`blueline` *destination method* **&rest** *keys* **&key** *layout hunt-bell working-bell figures place-notation place-bells* [Function]

Draws the blue line of *method* as a Scalable Vector Graphics (SVG) image. The *method* should have its stage and place notation set. While Roan only writes SVG format images, many other pieces of software, such as ImageMagick (`https://imagemagick.org/`), are able to convert SVG images to other formats.

The *destination* can be

- A text stream, open for writing: the SVG will be written to this stream, and the stream is returned as the value of the call to `blueline`.
- The symbol `t`: the SVG will be written to `*standard-output*`, and the value of `*standard-output*` is returned as the value of the call to `blueline`.
- A pathname: an SVG file will be written to this pathname, which will be opened with `if-exists :supersede`, and the truename of the resulting file is returned.
- A string with a fill pointer: the SVG will be appended to this string, as by `vector-push-extend`, which is returned.
- The symbol `nil`: the SVG will be written to a new string, which is returned.

Several keyword parameters can be used to control details of the image produced

layout Controls the distribution of leads into columns. For differentials, or methods with multiple, equal length cycles of working bells, each cycle always starts a new column. Within a cycle the value of *layout* controls the number of leads in a column. If it is a non-negative integer, this is the maximum number of rows in a column; though if the lead length exceeds this value each column will contain one lead. If `nil` this is no limit to

the number of leads in a column, each cycle of working bells then filling a column.The special value `:grid` may also be supplied, in which case only a single column is used for a single lead, with all the bells blue lines combined into it as a grid. The default value for *layout* is `100`.

hunt-bell Controls which hunt bells are displayed specially. Those not displayed specially, are treated as working bells. If a `bell`, that is, a small, non-negative integer less than the stage of *method*, this is the hunt bell displayed specially; a list of `bell`s may also be supplied, for multiple hunt bells. If a supplied `bell` is not actually a hunt bell of *method* it is ignored. The keyword `first` is equivalent to supplying whatever the smallest hunt bell of *method* is. The keyword `:all` is equivalent to supplying a list of all the hunt bells of *method*. The keyword `:working` treats all of the hunt bells as working bells. If *hunt-bell* is `nil` no hunt bells are displayed. The default value for *hunt-bell* is `:first`.

working-bell
Controls which working bell of each cycle is drawn first, the others following on in the order in which they are rung. This can be a `bell`, or a list thereof, or one of the keywords `:natural`, `:largest` or `:smallest`. If `:natural` for each cycle the largest bell that makes a place across the lead end is chosen; if there is no such bell in a cycle the largest bell in that cycle is used. For methods with Grandsire-like palindromic symmetry the first row of the lead is used instead of the lead end. The default value for *working-bell* is `:natural`.

figures If non-null figures will also be drawn, in addition to the blue line. If `t` they will be drawn for all leads. If `:lead` only for the first lead of each cycle. If `:half` and the *method* has the usual palindromic symmetry around the half lead, with one additional change at the lead end, they will only be drawn for the first half-lead; otherwise `:half` is equivalent to `:lead`. If `:head` the figures will only be drawn for the first lead head in each column. The default value for *figures* is `nil`.

place-notation
if non-null the place notation will be drawn to the left of the blue lines. If `t` it will be drawn for the first lead in each column. If `:lead` it will only be drawn for the first columnn. If `:half` and the *method* has the usual palindromic symmetry around the half lead, with one additional change at the lead end, it will only be drawn for the first half lead, plus at the lead end; otherwise `:half` is equivalent to `:lead`. The default value for `place-notation` is `nil`.

place-bells May have a value of `nil`, `:dot` or `:label`. If non-null dots are drawn where each place bell starts, and if `:label` a label is drawn to the right of the blue line at each place bell's start. The default value for *place-bells* is `:label`.

For an example, execute something like the following, and open the resulting file in a browser:

```
(blueline #P"/tmp/bastow.svg"
          (lookup-method-by-title "Bastow Little Bob Minor")
          :layout 12
          :figures :lead
          :place-notation :half)
```

Default values for the keyword arguments to this function can be set by assigning a property list of keywords and values to the variable `*blueline-default-parameters*`.

```
(equal
  (blueline nil
            (lookup-method-by-title "Advent Surprise Major")
            :layout nil
            :figures t
            :place-notation :lead)
  (let ((*blueline-default-parameters*
           '(:layout nil :figures t :place-notation :lead)))
    (blueline nil (lookup-method-by-title "Advent Surprise Major"))))
  ⇒ t
```

Signals a `type-error` if *destination* is not a stream, pathname, string with a fill pointer or one of the symbols `t` or `nil`; if *method* is not a `method`; if *layout* is not non-negative integer, `nil` or the keyword `:grid`; if *hunt-bell* is not a `bell`, list of bells, `nil` or one one of the keywords `:first`, `:all` or `:working`. if *working-bell* is not a `bell`, list of bells, or one of the symbols `:natural`, `:largest` or `smallest`; if *figures* is not one of the keywords `:none`, `:head`, `:half`, `:lead` or `:always`; if *place-notation* is not one of the keywords `:none`, `:half`, `:lead` or `:always`; or if *place-bells* is not `nil` or one of the keywords `:dot` or`:label`. Signals a `no-place-notation-error` if *method* doesn't have both its stage and place notation set. Can signal various errors if an I/O error occurs trying to write to a stream or create a file.

6 Internal 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 Peter Niblett's XML format documentation (`http://www.methods.org.uk/method-collections/xml-zip-files/method%20xml%201.0.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 conventionally 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 0æ*m*<*s*-1 and 0æ*n*<*s*-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"
```

7 Calls

Roan provides an immutable `call` object that describes a change ringing call, such as a bob or single, that modifies a lead of a `method`. A `call` usually has a fragment of place notation representing changes that are added to the the sequence of changes constituting the lead, typically replacing some existing changes in the lead.

A `call` has an offset, which specifies where in the lead the changes are added, replaced or deleted; this offset can be indexed from the beginning or the end of a lead, which frequently allows the same call to be used for similar methods with possibly different lead lengths. It is also possible to index from a postion within the lead rather than the beginning or end by supplying a fraction; again, this allows using, for example, half-lead calls with similar methods with different lead lengths.

Typically a `call` replaces exactly as many changes as it supplies. However it is possible to replace none, in which case the `call` adds to the lead length; to only replace changes with a zero length set of changes, in which case the `call` shortens the lead by deleting changes; or even to add more or fewer changes than it replaces.

Typically a call only affects the lead of a method to which is is applied. In exceptional cases, most notably doubles variations, it may also affect the subsequent lead. To support such use a `call` may have a following place notation fragment and a following replacement length. Such use is always restricted to being positioned at the beginning of the subsequent lead, and in the main lead the call must replace changes all the way to the end of the lead. Note that by starting the call at the end of the lead this could be simply adding changes, or even doing nothing.

A `call` is applied to a lead with the function `call-apply`. This can take multiple `call`s, all of which are applied to the same lead. They must not, however, overlap. The `call-apply` function returns two values. The first is a list of the changes of the lead, modified by the `call`(s). The second, if not `nil`, is another `call` to be applied to the following lead, and is only non-nil when a `call` does apply also to the subsequent lead.

Two `call`s may be compared with `equalp`.

Examples of `call`s:

- The usual bob for Cambridge Surprise is `(call "4")`.
- The usual single for Grandsire is `(call "3.123" :offset 2)`.
- The usual bob for Erin Triples is `(call "7" :from-end nil)`.
- A 58 half-lead bob for Bristol Major is `(call "5" :fraction 1/2)`.
- A bob in April Day Doubles is `(call "3.123" :following "3")`.
- A call for surprise that shortens the lead by omitting the first two blows, so that ringing of the lead commences at the backstroke snap is `(call nil :from-end nil :replace 2)`.

`call` [Type]

An immutable object describing a change ringing call, such as a bob or single.

`call` *place-notation* **&key** *from-end offset fraction replace following following-replace* [Function]

Creates and returns a `call`, which modifies the changes of a lead of a `method`. The *place-notation* argument is a string of place, the changes corresponding to which will

add or replace changes in a a lead of the `method` when applying the `code`. The *place-notation* may be `nil`, in which case no changes are add or replace existing ones. The *offset*, a non-negative integer, is the position at which to begin modifying the lead, and is measured from the beginning of the lead if the generalized boolean *from-end* is false, and from the end, otherwise. This can be further modifed by *fraction* which is multiplied by the lead length; the offset is counted forward or backward from that product. The `fraction`, if non-nill, must be a ratio greater than `0` and less than `1`, whose denominator evenly divides the lead length. The non-negative integer *replace* is the number of changes in the lead to be deleted or replaced. It is typically equal to the length of *changes*, which results in exact replacement of changes in the lead, but may be greater or less than that length, in which case the resulting lead is of a different length than a plain lead.

If either or both of *following* or *following-replace* are supplied the call is intended to also apply to the subsequent lead. These operate just like *place-notation* and *replace*, but on the subsequent lead, and always at the begining of that lead. This use also depends upon the caller of `call-apply` making correct use of its second return value.

If *replace* is not supplied or is `nil` it defaults to the number of changes represented by the *place-notation*. If *offset* is not supplied or is `nil`, it defaults to `0` if *from-end* is false, and otherwise to the value of *replace*, which may itself have been defaulted from the value of *place-notation*. The default value of *from-end* is `t`. The default value of *fraction* is `nil`. If *following* is supplied but *following-replace* is not, *following-replace* defaults to the number of changes represetned by *following*. If *following-replace* is supplied but *following* is not, *following* defaults to `nil`.

A `parse-error` is signaled if either *place-notation* or *following* is non-`nil` but not interpretable as place notation at the stage of *method*. A `type-error` is signaled if *offset* is supplied and is neither `nil` nor a non-negative integer; if *replace* is supplied and is neither `nil` nor a non-negative integer; *fraction* is supplied and is neither `nil` nor a ratio between `0` and `1`, exclusive; or if *following-replace* is supplied and is neither `nil` nor a non-negative integer.

`call-apply` *method* **&rest** *calls* [Function]

Applies zero or more *calls* to a lead of *method*. Returns two values, the first a list of `row`s constituting the changes of the modified lead and the second `nil` or a `call`, such that the call should be applied to the succeeding lead. This second value is only non-nil for complex calls that affect two consecutive leads, as are encountered in doubles variations. One or more of the *calls* may be `nil`, in which case they are ignored, just as if they had not been supplied. If no non-nil *calls* are supplied returns a list of the changes constituting a plain lead of *method*.

When multiple *calls* are supplied the indices of all are computed relative to the length and position within the plain lead, before the application of any others of the calls. For example, a half-lead call that replaces the 7th's in Cambridge Major continues to replace that change even if an earlier call removes or adds several changes.

Signals a `type-error` if *method* is not a `method` or if any of the *calls* are neither a `call` nor `nil`. Signals a `parse-error` if *method* does not have its stage or place-notation defined. Signals a `call-application-error` in any of the following circumstances: if the stage of *method* is such that the place notation or following place notation of one

or more of the *calls* is inapplicable; if an attempt is made to apply a fractional lead `call` where the denominator of the fraction does not evenly divide the lead length; if the `call` would be positioned, or replace changes, that lie outside the lead; if a `call` with following changes does not replace changes up to the end of the first lead, or an attempt is made to applly two or more `call`s with following place notation to the same lead.

`call-application-error` [Type]

Signaled when an anaomalous condition is detected while trying to apply a `call` to a `method`. Contains three potentially useful slots accessible with `call-application-error-call`, `call-application-error-method` and `call-application-error-details`.

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 suitably permissive licenses; however, you should read their actual licenses to be sure. See [dependencies], page 56.

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 domain or offered under a permissive, open source license, some are offered with a copyleft license. 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]
- Iterate (`https://common-lisp.net/project/iterate`) [MIT]
- CL-INTERPOL (`https://edicl.github.io/cl-interpol/`) [BSD]
- ASDF (`https://common-lisp.net/project/asdf/`) [MIT]
- CL-FAD (`http://weitz.de/cl-fad/`) [BSD]
- local-time (`https://common-lisp.net/project/local-time/`) [MIT]
- uuid (`https://github.com/dardoria/uuid/blob/master/uuid.lisp`) [LLGPL]
- CL-PPCRE (`http://weitz.de/cl-ppcre`) [BSD]
- Drakma (`http://weitz.de/drakma/`) [BSD]
- Plump (`https://shinmera.github.io/plump/`) [Artistic]
- binascii (`https://github.com/froydnj/binascii`) [BSD]
- ZIP (`https://common-lisp.net/project/zip/`) [BSD+LGPL]

Note that several of these libraries in turn load others, if embedding them in something you ship you may need to pay attention to the licenses of things transitively included.

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 59) 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 58) 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]

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 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.

C.1 What's with the name?

Robert Roan was an important seventeenth century ringer and composer. He is widely 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 less than a minute 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.

One caveat: a few of the tests, related to upgrading the methods library, require access to the internet. If it is not available, there will be a failures reported.

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

V

W

X