slippery chicken

Generated with ROBODoc Version 4.99.43 (Oct 1 2016)

October 27, 2020

Contents

1   sc/all.lsp

[Modules]

NAME:

 all

```
File:             all.lsp

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Load all the lisp files associated with slippery-chicken.
                  No public interface envisaged (so no robodoc entries).

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    5th December 2000

$$ Last modified:  19:16:27 Tue Jul 14 2020 CEST
```

2   sc/clm-akoustik-piano-spectra.lsp

[Modules]

NAME:

 globals

```
File:             clm-piano-spectra.lsp

Class Hierarchy:  None: no classes defined.

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          For reading in from spectra.lsp: From Akoustic Piano's
                  samples library: "/Akoustik Piano
                  Library/Samples/ConcertGrand/Samples/". We can use this
                  data for the calculation of chord dissonance values in
                  the chord class's calculate-dissonance method.
```

```
Author:         Michael Edwards: m@michael-edwards.org

Creation date:  5th August 2015

$$ Last modified: 12:30:03 Tue Feb  2 2016 GMT

SVN ID: $Id: clm-piano-spectra.lsp 963 2010-04-08 20:58:32Z medward2 $
```

3   sc/clm-piano-spectra.lsp

[Modules]
NAME:

```
 globals

 File:           clm-piano-spectra.lsp

 Class Hierarchy: None: no classes defined.

 Version:        1.0.11

 Project:        slippery chicken (algorithmic composition)

 Purpose:        For reading in from spectra.lsp: Modified data from CLM's
                 spectr.clm which itself was created from data provided
                 courtesy of J.A. Moorer. We'll use this data for the
                 calculation of chord dissonance values in the chord
                 class's calculate-dissonance method.

 Author:         Michael Edwards: m@michael-edwards.org

 Creation date:  5th August 2015

 $$ Last modified: 16:22:57 Wed Aug  5 2015 BST

 SVN ID: $Id: clm-piano-spectra.lsp 963 2010-04-08 20:58:32Z medward2 $
```

4   sc/clm-violin-ensemble-spectra.lsp

[Modules]
NAME:

```
globals

File:              violin-ensemble-spectra.lsp

Class Hierarchy:   None: no classes defined.

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           For reading in from spectra.lsp: From Kontakt's violin
                   ensemble samples library: Kontakt 3 Library/Orchestral/Z
                   - Samples/01 Violin ensemble - 14/VI-14_mV_sus_mf". We can
                   use this data for the calculation of chord dissonance
                   values in the chord class's calculate-dissonance method.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     5th August 2015

$$ Last modified:  19:36:36 Thu Jun  8 2017 BST

SVN ID: $Id: clm-piano-spectra.lsp 963 2010-04-08 20:58:32Z medward2 $
```

5   sc/cm

[Modules]
NAME:

```
cm

File:              cm.lsp

Class Hierarchy:   none (no classes defined)

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Definition of common-music related and other functions
                   like transposition of notes/chords, enharmonic
                   equivalents etc.

Author:            Michael Edwards: m@michael-edwards.org
```

```
Creation date:    1st March 2001

$$ Last modified:  08:37:26 Thu Mar 26 2020 CET

SVN ID: $Id$
```

5.1   cm/degree-to-note

[cm] [Functions]

DESCRIPTION:

```
Get the specified scale degree number as a note-name pitch symbol within
the current scale. An optional argument allows the user to specify that the
scale degree number should be used to get the note-name pitch from a
different scale.
```

ARGUMENTS:

```
An integer that is a scale degree number.
```

OPTIONAL ARGUMENTS:

```
- The scale from which the note-name pitch symbol associated with the
  specified scale degree is to be drawn.
```

RETURN VALUE:

```
A note-name pitch symbol.
```

EXAMPLE:

```
(degree-to-note 127 'chromatic-scale)

=> G9

(degree-to-note 127 'twelfth-tone)

=> ATS0

(degree-to-note 127 'quarter-tone)

=> EQF4
```

SYNOPSIS:

```
(defun degree-to-note (degree &optional (scale cm::*scale*))
```

5.2   cm/degrees-per-octave

[cm] [Functions]

DESCRIPTION:

Return the number of scale degrees in the span of one octave within the
current tuning system.

ARGUMENTS:

 - No arguments.

RETURN VALUE:

 - An integer that is the number of scale degrees in each octave.

EXAMPLE:

```
(in-scale :chromatic)
(degrees-per-octave)

=> 12

(in-scale :quarter-tone)
(degrees-per-octave)

=> 24
```

SYNOPSIS:

```
(defun degrees-per-octave (&optional (scale cm::*scale*))
```

5.3   cm/degrees-per-semitone

[cm] [Functions]

DESCRIPTION:

Get the number of scale degrees per equal-tempered semitone in the current
tuning scale.

ARGUMENTS:

- No arguments

OPTIONAL ARGUMENTS:

- The scale for which the number of degrees per semitone is to be
  retrieved.

RETURN VALUE:

 An integer.

EXAMPLE:

(degrees-per-semitone 'chromatic-scale)

=> 1

(degrees-per-semitone 'twelfth-tone)

=> 6

(degrees-per-semitone 'quarter-tone)

=> 2

SYNOPSIS:

(defun degrees-per-semitone (&optional (scale cm::*scale*))

5.4   cm/degrees-to-notes

[cm] [Functions]

DESCRIPTION:

 NB: If the specified scale-degree number within the current scale would
    result in pitch outside of the maximum MIDI pitch range for that tuning
    (chromatic: C-1 to B10; quarter-tone: C-1 to BQS10; twelfth-tone: C-1
    to CTF11), the function will return an error.

ARGUMENTS:

 An integer that is a scale degree number in the current tuning.

RETURN VALUE:

 A list of note-name pitch symbols.

EXAMPLE:

```
(in-scale :chromatic)
(degrees-to-notes '(0 143 116 127 38))

=> (C-1 B10 AF8 G9 D2)

(in-scale :twelfth-tone)
(degrees-to-notes '(0 144 116 127 38 287 863))

=> (C-1 C1 GSS0 ATS0 FSSS-1 CTF3 CTF11)

(in-scale :quarter-tone)
(degrees-to-notes '(0 144 116 127 38 287))

=> (C-1 C5 BF3 EQF4 G0 BQS10)
```

SYNOPSIS:

```
(defun degrees-to-notes (degrees)
```

5.5 cm/event-list-to-midi-file

[cm] [Functions]

DESCRIPTION:

 Write the events in a list to a midi-file. events-update-time is a related
 function useful for preparind ad-hoc events for midi-file-writing.

ARGUMENTS:

 - A list of events objects
 - the path to the midi-file
 - the starting tempo (integer: BPM)
 - a time-offset for the events (seconds)

OPTIONAL ARGUMENTS:

 - whether to overwrite events' amplitude slots and use a single
 velocity/amplitude value given here (0-1.0 (float) or 0-127 (integer)

RETURN VALUE:

 The MIDI file path

SYNOPSIS:

```
(defun event-list-to-midi-file (event-list
                                 &key (midi-file "/tmp/tmp.mid")
                                   (start-tempo 120) (time-offset 0)
                                   (auto-open (get-sc-config
                                                 'midi-play-auto-open))
                                 force-velocity)
```

5.6 cm/freq-to-degree

[cm] [Functions]

DESCRIPTION:

 Get the scale degree of the specified frequency in Hertz within the current
 scale.

 NB: This method will return fractional scale degrees.

ARGUMENTS:

 A frequency in Hertz.

OPTIONAL ARGUMENTS:

 - The scale in which to find the corresponding scale degree.

RETURN VALUE:

 A scale degree number. This may be a decimal number.

EXAMPLE:

```
(freq-to-degree 423 'chromatic-scale)
```

```
=> 68.317856
```

```
(freq-to-degree 423 'twelfth-tone)
```

```
=> 409.9071

(freq-to-degree 423 'quarter-tone)

=> 136.63571
```

SYNOPSIS:

```
(defun freq-to-degree (freq &optional (scale cm::*scale*))
```

5.7   cm/freq-to-midi

[cm] [Functions]

DATE:

```
10th July 2016, Essen Werden
```

DESCRIPTION:

```
convert a frequency in Hertz to a MIDI note number (possibly floating
point, indicating microtonality).
```

ARGUMENTS:

```
a frequency in Hertz
```

RETURN VALUE:

```
a floating point value representing the MIDI note number of the given frequency
```

EXAMPLE:

```
(freq-to-midi 260) --> 59.892094
(freq-to-midi (midi-to-freq 60)) --> 60.0
```

SYNOPSIS:

```
(defun freq-to-midi (freq)
```

5.8   cm/freq-to-note

[cm] [Functions]

DESCRIPTION:

Get the note-name pitch equivalent of the specified frequency, rounded to
the nearest scale degree of the current scale.

ARGUMENTS:

A number that is a frequency in Hertz.

OPTIONAL ARGUMENTS:

- The scale in which the note-name pitch equivalent is to be sought (Common
Music scale object or symbol). If a symbol, then 'chromatic-scale,
'twelfth-tone, or 'quarter-tone only at present.

RETURN VALUE:

A note-name pitch symbol.

EXAMPLE:

(freq-to-note 423 'chromatic-scale)

=> AF4

(freq-to-note 423 'twelfth-tone)

=> GSSS4

(freq-to-note 423 'quarter-tone)

=> AQF4

SYNOPSIS:

(defun freq-to-note (freq &optional (scale cm::*scale*))

5.9   cm/get-pitch-bend

[cm] [Functions]

DESCRIPTION:

Get the MIDI pitch-bend value necessary for application to a MIDI pitch in
order to achieve the specified frequency.

NB: This will always return a positive value between 0.0 and 1.0, as
    slippery-chicken always applies pitch-bends upwards from the nearest
    chromatic note below the specified frequency.

NB: This value will be the same in all tuning scales.

ARGUMENTS:

 A frequency in Hertz.

RETURN VALUE:

 A two-digit decimal number that is the pitch-bend value required to achieve
 the specified frequency in MIDI.

EXAMPLE:

```
(get-pitch-bend 423)

=> 0.32
```

SYNOPSIS:

```
(defun get-pitch-bend (freq)
```

5.10   cm/in-scale

[cm] [Functions]

DESCRIPTION:

 Set the global scale (tuning) for the current slippery-chicken
 environment. Current options are :chromatic, :quarter-tone or
 :twelfth-tone. See the file cm-load.lsp for specifications and the html
 manual page "More about note-names and scales" for more details on use.

ARGUMENTS:

 - A scale (tuning) designation.

RETURN VALUE:

 the (new) Common Music scale object

EXAMPLE:

```
(in-scale :chromatic)

=> #<tuning "chromatic-scale">

(in-scale :quarter-tone)

=> #<tuning "quarter-tone">

(in-scale :twelfth-tone)

=> #<tuning "twelfth-tone">
```

SYNOPSIS:

```
(defun in-scale (scale)
```

5.11   cm/midi-file-high-low

[cm] [Functions]

DATE:

```
 30-Dec-2010
```

DESCRIPTION:

```
 Print the highest and lowest pitch in a specified MIDI file as a MIDI note
 number.
```

```
 NB: This is a Common Music function and as such must be called with the
     package qualifier cm:: if used within slippery chicken.
```

ARGUMENTS:

```
 - The path (including the name) to the MIDI file.
```

OPTIONAL ARGUMENTS:

```
 - An integer or NIL to indicate which track in the specified MIDI file is
   to be accessed. If NIL, all tracks will be accessed. NB: CM (and
   therefore slippery-chicken too) generates some MIDI files by writing each
   channel to a different track, so the "track" would seem synonymous with
   "channel" here.
```

RETURN VALUE:

 Two integer values (using the values function) that are the highest and
 lowest pitches in the specified MIDI file.

EXAMPLE:

(cm::midi-file-high-low "/tmp/multi-ps.mid")

=> 72, 60

SYNOPSIS:

(defun midi-file-high-low (file &optional track)

5.12   cm/midi-file-one-note

[cm] [Functions]

DESCRIPTION:

 Write all midi notes in the file out to a new one-channel file using the
 single pitch <note> and channel number <channel>.

ARGUMENTS:

 - A string that is the file path, including file-name and extension.
 - A note-name symbol or MIDI-note integer that is the pitch to write.
 - An integer that is the channel to which the output should be written
   (1-based)

OPTIONAL ARGUMENTS:

 - An integer that is the an existing MIDI channel in the original file. If
   this argument is given, only notes on this channel of the original file
   will be written (1-based).

RETURN VALUE:

 The path to the new file.

EXAMPLE:

(cm::midi-file-one-note "/tmp/multi-ps.mid" 'c4 1)

SYNOPSIS:

(defun midi-file-one-note (file note channel &optional old-channel)

5.13 cm/midi-file-to-events

[cm] [Functions]

DATE:

 28/6/16, Edinburgh

DESCRIPTION:

 Read in a MIDI file and convert notes to event objects.
 NB This won't (yet) import microtones as indicated by pitch-bend messages

ARGUMENTS:

 the path to the midi file

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :track. The track number to read. Default = NIL which means read all
 tracks.
 - :tempo. The tempo of the track in crotchets (quarter notes) per
 minute. Default, q = 120.

RETURN VALUE:

 a list of event objects

SYNOPSIS:

(defun midi-file-to-events (file &key track (tempo 120.0))

5.14 cm/midi-to-degree

[cm] [Functions]

DESCRIPTION:

 Convert the specified MIDI note number to the degree number of the current
 scale.

ARGUMENTS:

 - A MIDI note number.

RETURN VALUE:

 – An integer that is the scale-degree equivalent of the specified MIDI note
   number in the current scale.

EXAMPLE:

```
(in-scale :chromatic)
(midi-to-degree 64)

=> 64

(in-scale :twelfth-tone)
(midi-to-degree 64)

=> 384

(in-scale :quarter-tone)
(midi-to-degree 64)

=> 128
```

SYNOPSIS:

```
(defun midi-to-degree (midi-note &optional (scale cm::*scale*))
```

5.15   cm/midi-to-freq

[cm] [Functions]

DESCRIPTION:

 Get the frequency equivalent in Hertz to the specified MIDI note number.

ARGUMENTS:

 – A number (can be a decimal) that is a MIDI note number.

RETURN VALUE:

 A decimal number that is a frequency in Hertz.

EXAMPLE:

```
(midi-to-freq 67)
```

```
=> 391.99542
```

```
(midi-to-freq 67.9)
```

```
=> 412.91272
```

SYNOPSIS:

```
(defun midi-to-freq (midi-note)
```

5.16   cm/midi-to-note

[cm] [Functions]

DESCRIPTION:

```
 Get the note-name pitch symbol equivalent of the specified MIDI note
 number.
```

ARGUMENTS:

```
 - a MIDI note number: integer or float. If float, appropriate microtones
 will be returned if the current scale is microtonal (e.g. (in-scale
 :quarter-tone))
```

RETURN VALUE:

```
 A note-name pitch symbol.
```

EXAMPLE:

```
(midi-to-note 67)
=> G4
(midi-to-note 60.5)
=>CQS4
```

SYNOPSIS:

```
(defun midi-to-note (midi-note &optional (scale cm::*scale*))
```

5.17   cm/midi2qlist

[cm] [Functions]

DATE:

 10th November 2016, Edinburgh

DESCRIPTION:

 Convert a midi-file to qlist text format, for sequencing in PD or MaxMSP.
 If you don't want specific tracks from the file, just pass two arguments so
 <tracks> remains nil.

ARGUMENTS:

 - the path to the midi file you want to convert (string)
 - the path to the qlist text file you'd like to write (string). If this is
   nil then we'll write to the same patch as the midi file with the extension
   .txt added. NB If this file exists it will be overwritten, no questions
   asked.
 - (&rest) the numbers of the tracks you'd like to convert, starting from 1.

RETURN VALUE:

 The number of notes written (integer)

SYNOPSIS:

```
(defun midi2qlist (midi-file qlist-file &rest tracks)
```

5.18   cm/note-to-degree

[cm] [Functions]

DESCRIPTION:

 Get the scale degree number of the specified note-name pitch symbol within
 the current scale.

ARGUMENTS:

 - A note-name pitch symbol.

OPTIONAL ARGUMENTS:

- The scale in which to find the scale-degree of the specified pitch.

RETURN VALUE:

 An integer that is a scale degree in the current scale.

EXAMPLE:

```
(note-to-degree 'AF4 'chromatic-scale)
```

```
=> 68
```

```
(note-to-degree 'AF4 'twelfth-tone)
```

```
=> 408
```

```
(note-to-degree 'AF4 'quarter-tone)
```

```
=> 136
```

SYNOPSIS:

```
(defun note-to-degree (note &optional (scale cm::*scale*))
```

5.19   cm/note-to-freq

[cm] [Functions]

DESCRIPTION:

 Get the frequency in Hertz of the specified note-name pitch symbol.

ARGUMENTS:

 - A note-name pitch symbol.

RETURN VALUE:

 A frequency in Hertz.

EXAMPLE:

```
(in-scale :chromatic)
(note-to-freq 'AF4)
```

```
=> 415.3047

(in-scale :twelfth-tone)
(note-to-freq 'GSSS4)

=> 423.37845

(in-scale :quarter-tone)
(note-to-freq 'AQF4)

=> 427.47403
```

SYNOPSIS:

```
(defun note-to-freq (note)
```

5.20   cm/note-to-midi

[cm] [Functions]

DESCRIPTION:

```
 Get the MIDI note number equivalent for a chromatic note-name pitch
 symbol.
```

ARGUMENTS:

```
 - A chromatic note-name pitch symbol.
```

RETURN VALUE:

```
 An integer.
```

EXAMPLE:

```
(note-to-midi 'g4)

=> 67
```

SYNOPSIS:

```
(defun note-to-midi (midi-note)
```

5.21   cm/parse-midi-file

[cm] [Functions]

DESCRIPTION:

 Return Common Music MIDI events in the specified file as a list.

 NB: This is a Common Music function and as such must be called with the
     package qualifier cm:: if used within slippery chicken.

ARGUMENTS:

 - The path (including the file name) to the MIDI file.

OPTIONAL ARGUMENTS:

 - An integer or NIL to indicate which track in the specified MIDI file is
   to be accessed. If NIL, all tracks will be accessed. NB: CM (and
   therefore slippery-chicken too) generates some MIDI files by writing each
   channel to a different track, so the "track" would seem synonymous with
   "channel" here.

RETURN VALUE:

 The CM data for the MIDI events in the specified file, and the number of
 events.

EXAMPLE:

```
(cm::parse-midi-file "/tmp/multi-ps.mid")

=>
(#i(midi-tempo-change time 0.0 usecs 357142)
 #i(midi-time-signature time 0.0 numerator 2 denominator 4 clocks 24 32nds 8)
 #i(midi time 0.0 keynum 36 duration 0.357142 amplitude 0.09448819 channel 15)
 #i(midi-tempo-change time 0.0 usecs 357142)
 #i(midi-time-signature time 0.0 numerator 2 denominator 4 clocks 24 32nds 8)
 #i(midi-tempo-change time 0.0 usecs 357142)
 #i(midi time 0.178571 keynum 66 duration 0.178571 amplitude 0.09448819 channel 15)
 #i(midi time 0.357142 keynum 68 duration 0.0892855 amplitude 0.09448819 channel 15)
 #i(midi time 0.357142 keynum 40 duration 0.357142 amplitude 0.09448819 channel 15)
 #i(midi time 0.6249985 keynum 66 duration 0.0892855 amplitude 0.09448819 channel 15)
 #i(midi-time-signature time 0.714284 numerator 3 denominator 4 clocks 24 32nds 8)
```

SYNOPSIS:

```
(defun parse-midi-file (file &optional track)
```

6   sc/cm-cm

[Modules]

NAME:

```
cm

 File:             cm-cm.lsp

 Class Hierarchy:  none (no classes defined)

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Definition of common-music functions etc. that need to be
                   in the cm package

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    26th March 2020 (abstracted out of cm.lsp)

 $$ Last modified:  08:37:09 Thu Mar 26 2020 CET

 SVN ID: $Id$
```

7   sc/cm-load

[Modules]

NAME:

```
cm-load

 File:             cm-load.lsp

 Class Hierarchy:  none (no classes defined)

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Definition of the common-music quarter-tone scale and
                   twelfth-tone scale which should be loaded and not
```

```
                        compiled.  The quarter tone scale is our default
                        No public interface envisaged (so no robodoc entries).

 Author:                Michael Edwards: m@michael-edwards.org

 Creation date:         7th February 2003

 $$ Last modified:      12:21:03 Mon May 28 2018 CEST

 SVN ID: $Id$
```

8   sc/cmn

[Modules]

NAME:

```
 cmn

 File:                  cmn.lsp

 Class Hierarchy:       None: no classes defined.

 Version:               1.0.11

 Project:               slippery chicken (algorithmic composition)

 Purpose:               Interface from complete-set to Bill's CMN package for
                        displaying of sets in musical notation.

 Author:                Michael Edwards: m@michael-edwards.org

 Creation date:         11th February 2002

 $$ Last modified:      08:42:38 Thu Mar 26 2020 CET

 SVN ID: $Id$
```

9   sc/cmn-glyphs

[Modules]

NAME:

```
cmn-glyphs

File:              cmn-glyphs.lsp

Class Hierarchy:   none, no classes defined

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Definition of various postscript glyphs (accidentals
                   etc.) for cmn.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     10th November 2002

$$ Last modified:  17:54:01 Mon May 28 2018 CEST

SVN ID: $Id$
```

10   sc/get-spectrum

[Modules]

NAME:

```
get-spectrum

File:              get-spectrum.lsp

Class Hierarchy:   none (no classes defined)

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Used mainly by spectra.lsp, get-spectrum returns, as a
                   list, the most prominent frequencies in a sound file.
                   The list is ordered from the most prominent to the least
                   prominent frequency (:order-by 'amp) or the highest to
                   lowest frequency (:order-by 'freq) and the number of
                   elements in the list is set by the <num-partials>
                   parameter.  The instrument spec-an is called by
                   get-spectrum and is therefore not meant to be used as a
```

```
                       public function (it is written as an instrument to take
                       advantage of CLM's run-loop efficiency).  The results of
                       analysis are stored in
                       *slippery-chicken-get-spectrum-last-result* so that if
                       you request the same analysis repeatedly, the analysis
                       will not need to be reperformed each time.  Be careful
                       though, only the input file, the ordering, the number of
                       partials requested and the analysis start time are used
                       to detect whether analysis has to be reperformed or not.
                       If you want to force reanalysis, do (setf
                       *slippery-chicken-get-spectrum-last-result* nil) before
                       the call to get-spectrum.

 Creation date:        This is very old code but it was added to slippery
                       chicken on August 5th 2015

 Author:               Michael Edwards: m@michael-edwards.org

 $$ Last modified:  10:21:02 Tue Aug 28 2018 CEST

 SVN ID: $Id: get-spectrum.lsp 5359 2015-07-24 20:53:22Z medward2 $
```

11   sc/globals.lsp

[Modules]

NAME:

```
 globals

 File:                 globals.lsp

 Class Hierarchy:      None: no classes defined.

 Version:              1.0.11

 Project:              slippery chicken (algorithmic composition)

 Purpose:              Definition of the user-changeable configuration data and
                       globals for internal programme use. See top of cmn.lsp and
                       osc-sc-bsd.lsp for a few more globals relevant to
                       functionality in their respective packages.

 Author:               Michael Edwards: m@michael-edwards.org
```

```
Creation date:     30th May 2013

$$ Last modified:  16:28:22 Thu Oct  8 2020 CEST

SVN ID: $Id: sclist.lsp 963 2010-04-08 20:58:32Z medward2 $
```

12   sc/instruments

[Modules]

NAME:

```
instrument

File:              instruments.lsp

Class Hierarchy:   none (no classes defined)

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Definition of various standard instruments and other
                   data/functions useful to slippery-chicken users.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     30th December 2010

$$ Last modified:  17:13:44 Thu Oct  8 2020 CEST

SVN ID: $Id$
```

12.1   instruments/+slippery-chicken-standard-instrument-palette+

[instruments] [Global Parameters]

DESCRIPTION:

```
A palette of standard instruments (by no means exhaustive...) for use
directly in projects or for combining with user palettes e.g.

(combine
 +slippery-chicken-standard-instrument-palette+
```

```
  (make-instrument-palette
   'esoteric-stuff
    '((toy-piano
        (:staff-name "toy piano" ....
```

SYNOPSIS:

```
(defparameter +slippery-chicken-standard-instrument-palette+
  (make-instrument-palette
   'slippery-chicken-standard-instrument-palette
   ;; SAR Fri Jan 20 11:43:32 GMT 2012: Re-ordering these to Adler's "standard"
   ;; score order for easier look-up
   '((piccolo
       (:staff-name "piccolo" :staff-short-name "picc"
        :lowest-written d4 :highest-written c7 :transposition-semitones 12
        :missing-notes nil
        :largest-fast-leap 19
        :starting-clef treble
        :chords nil
        :microtones t
        :midi-program 73))
      (flute
       (:staff-name "flute" :staff-short-name "fl"
        :lowest-written c4 :highest-written d7
        :missing-notes (cqs4 dqf4)
        :largest-fast-leap 19
        :starting-clef treble
        :chords nil
        :microtones t
        :midi-program 74))
      (alto-flute
       (:staff-name "alto flute" :staff-short-name "alt fl"
        :lowest-written c4 :highest-written c7 :transposition-semitones -5
        :missing-notes (cqs4 dqf4)
        :largest-fast-leap 17
        :starting-clef treble
        :chords nil
        :microtones t
        :midi-program 74))
      ;; SAR Fri Jan 20 11:46:45 GMT 2012: Modified bass flute range to that
      ;; stated by Adler.
      (bass-flute
       (:staff-name "bass flute" :staff-short-name "bass fl"
        :lowest-written c4 :highest-written c7 :transposition-semitones -12
        :missing-notes (cqs4 dqf4)
        :largest-fast-leap 15
```

```
      :clefs-in-c (treble bass) :starting-clef treble
      :chords nil
      :microtones t
      :midi-program 74))
;; SAR Fri Jan 20 12:01:37 GMT 2012: Added oboe. Conservative range taken
;; from the Adler
(oboe
 (:staff-name "oboe" :staff-short-name "ob"
  :lowest-written bf3 :highest-written a6
  :largest-fast-leap 12
  :starting-clef treble
  :chords nil :microtones t
  :missing-notes (bqf3 bqs3 cqs4 dqf4)
  :midi-program 69))
;; MDE Tue Feb 20 19:43:50 2018 --
(cor-anglais
 (:staff-name "cor anglais" :staff-short-name "c.a."
  :lowest-written bf3 :highest-written a6
  :transposition-semitones -7
  :largest-fast-leap 12
  :starting-clef treble
  :chords nil :microtones t
  :missing-notes (bqf3 bqs3 cqs4 dqf4)
  :midi-program 70))
(oboe-d-amore
 (:staff-name "oboe d'amore" :staff-short-name "ob d'am"
  :lowest-written bf3 :highest-written a6
  :transposition-semitones -3
  :largest-fast-leap 12
  :starting-clef treble
  :chords nil :microtones t
  :missing-notes (bqf3 bqs3 cqs4 dqf4)
  :midi-program 70))
(e-flat-clarinet
 (:staff-name "E-flat clarinet" :staff-short-name "E-flat cl"
  :lowest-written e3 :highest-written a6 :transposition-semitones 3
  :missing-notes (aqs4 bqf4 bqs4 cqs5 dqf5 gqf3 fqs3 fqf3)
  :largest-fast-leap 15
  :starting-clef treble
  :chords nil
  :microtones t
  :midi-program 72))
(b-flat-clarinet
 (:staff-name "B-flat clarinet" :staff-short-name "B-flat cl"
  :lowest-written e3 :highest-written a6 :transposition-semitones -2
  :missing-notes (aqs4 bqf4 bqs4 cqs5 dqf5 gqf3 fqs3 fqf3)
```

```
      :largest-fast-leap 15
      :starting-clef treble
      :chords nil
      :microtones t
      :midi-program 72))
   (a-clarinet
    (:staff-name "A clarinet" :staff-short-name "A cl"
     :lowest-written e3 :highest-written a6 :transposition-semitones -3
     :missing-notes (aqs4 bqf4 bqs4 cqs5 dqf5 gqf3 fqs3 fqf3)
     :largest-fast-leap 15
     :starting-clef treble
     :chords nil
     :microtones t
     :midi-program 72))
   (bass-clarinet
    (:staff-name "bass clarinet" :staff-short-name "bass cl"
     :lowest-written c3 :highest-written g6 :transposition-semitones -14
     :missing-notes (aqs4 bqf4 bqs4 cqs5 dqf5 gqf3 fqs3 fqf3 eqf3 dqs3 dqf3
                       cqs3)
     :largest-fast-leap 13
     :prefers-notes low
     :clefs (treble) :clefs-in-c (treble bass) :starting-clef treble
     :chords nil
     :microtones t
     :midi-program 72))
   (soprano-sax
    (:staff-name "soprano saxophone" :staff-short-name "sop sax"
     :lowest-written bf3 :highest-written fs6 :transposition-semitones -2
     :missing-notes (gqs4 gqs5)
     :largest-fast-leap 15
     :starting-clef treble
     :chords nil
     :microtones t
     :midi-program 65))
   (alto-sax
    (:staff-name "alto saxophone" :staff-short-name "alt sax"
     ;; altissimo extra....by hand...
     :lowest-written bf3 :highest-written fs6 :transposition-semitones -9
     :missing-notes (gqs4 gqs5)
     :largest-fast-leap 15
     :starting-clef treble
     :chords nil
     :microtones t
     :midi-program 66))
   (tenor-sax
    (:staff-name "tenor sax" :staff-short-name "ten sax"
```

```
      :lowest-written bf3 :highest-written fs6 :transposition-semitones -14
      :missing-notes (gqs4 gqs5)
      :largest-fast-leap 13
      :starting-clef treble :clefs-in-c (treble bass)
      :chords nil
      :microtones t
      :midi-program 67))
    (baritone-sax
     (:staff-name "baritone sax" :staff-short-name "bar sax"
      :lowest-written bf3 :highest-written fs6 :transposition-semitones -21
      :missing-notes (gqs4 gqs5)
      :largest-fast-leap 11
      :clefs-in-c (treble bass) :starting-clef treble
      :chords nil
      :microtones t
      :midi-program 68))
    (bassoon
     (:staff-name "bassoon" :staff-short-name "bsn"
      ;; of course it can go higher but best not to algorithmically select
      ;; these
      :lowest-written bf1 :highest-written c5
      ;; Wolfgang Ruediger says all 1/4 tones are OK above low E
      :missing-notes (bqf1 bqs1 cqs2 dqf2 dqs2 eqf2)
      :largest-fast-leap 13
      :clefs (bass tenor) :starting-clef bass
      :chords nil :microtones t :midi-program 71))
    (contra-bassoon
     (:staff-name "contrabassoon" :staff-short-name "cbsn"
      ;; some pieces go up to c5 but let's not
      :lowest-written bf1 :highest-written a4
      :transposition-semitones -12
      ;; Wolfgang Ruediger says all 1/4 tones are OK above low E for the
      ;; bassoon, so let's assume for now that these work here too.
      :missing-notes (bqf1 bqs1 cqs2 dqf2 dqs2 eqf2)
      :largest-fast-leap 13
      :clefs (bass tenor) :starting-clef bass
      :chords nil :microtones t :midi-program 71))
    (french-horn
     (:staff-name "french horn" :staff-short-name "hn"
      :lowest-written c3 :highest-written c6 :transposition-semitones -7
      :largest-fast-leap 9
      ;; :clefs (treble bass) :starting-clef treble
      :clefs (treble)
      :chords nil
      :microtones t
      :midi-program 61))
```

```
(c-trumpet
 (:staff-name "trumpet in c" :staff-short-name "c tpt"
  :lowest-written fs3 :highest-written c6
  :largest-fast-leap 9
  :clefs (treble) :starting-clef treble
  :chords nil
  :microtones t
  :midi-program 57))
;; SAR Fri Jan 20 12:09:41 GMT 2012: Added b-flat-trumpet from Adler
;; MDE Mon Feb 20 20:02:55 2012 -- modified to keep in line with clarinet
(b-flat-trumpet
 (:staff-name "B-flat trumpet" :staff-short-name "b-flat tpt"
  ;; the -flat should be converted in CMN and LilyPond to the flat sign
  :lowest-written fs3 :highest-written d6 :transposition-semitones -2
  :largest-fast-leap 9
  :starting-clef treble
  :chords nil :midi-program 57))
;; SAR Fri Jan 20 12:17:24 GMT 2012: Added tenor trombone from Adler
(tenor-trombone
 (:staff-name "trombone" :staff-short-name "tbn"
  :lowest-written e2 :highest-written bf4
  :largest-fast-leap 7
  :microtones t
  :clefs (bass tenor) :starting-clef bass
  :chords nil :midi-program 58))
;; MDE Thu Nov  8 19:32:19 2018 --
(bass-trombone
 (:staff-name "bass trombone" :staff-short-name "btbn"
  ;; these are of course not true for every instrument/player
  :lowest-written e1 :highest-written g4
  :largest-fast-leap 7
  :microtones t
  :clefs (bass tenor) :starting-clef bass
  :chords nil :midi-program 58))
;;; SAR Fri Jul 13 12:35:35 BST 2012
(tuba
 (:staff-name "tuba" :staff-short-name "tba"
  :lowest-written d1 :highest-written g4
  :largest-fast-leap 5
  :clefs (bass) :starting-clef bass
  :chords nil :midi-program 59))
(marimba
 (:staff-name "marimba" :staff-short-name "mba"
  :lowest-written c3 :highest-written c7
  :starting-clef treble :clefs (treble) ; (treble bass)
  :chords t :chord-function mallet-chord-fun
```

```
      :microtones nil :midi-program 13))
    (vibraphone
     (:staff-name "vibraphone" :staff-short-name "vib"
      :lowest-written f3 :highest-written f6
      :starting-clef treble
      :chords t :chord-function mallet-chord-fun
      :microtones nil :midi-program 12))
    (accordion
     (:staff-name "accordion" :staff-short-name "acc"
      :lowest-written e1 :highest-written bf7
      :clefs (bass treble) :starting-clef treble
      ;; clearly a better function should be provided at some point
      :chords t :chord-function piano-chord-fun
      :microtones nil :midi-program 22))
    ;; MDE Mon Mar 24 20:21:08 2014 -- following three added from data given
    ;; by Zach Howell (thanks).
    (glockenspiel
     (:staff-name "glockenspiel" :staff-short-name "glk"
                  :lowest-written f3 :highest-written c6
                  :transposition-semitones +24
                  :starting-clef treble
                  :chords nil :microtones nil :missing-notes nil
                  :midi-program 10))
    (xylophone
     (:staff-name "xylophone" :staff-short-name "xyl"
                  :lowest-written f3 :highest-written c7
                  :transposition-semitones +12
                  :starting-clef treble
                  :chords nil :microtones nil :missing-notes nil
                  :midi-program 14))
    (celesta
     (:staff-name "celesta" :staff-short-name "cel"
                  :lowest-written c3 :highest-written c7
                  :transposition-semitones +12
                  :starting-clef treble
                  :chords t :microtones nil :missing-notes nil
       :midi-program 9))
    (crotales
     (:staff-name "crotales" :staff-short-name "crt"
      :lowest-written c4 :highest-written c6
      :starting-clef treble
      :transposition-semitones 24
      :chords t :chord-function mallet-chord-fun
      ;; no crotales in GM set so use glockenspiel
      :microtones nil :midi-program 10))
    (piano
```

```
   (:staff-name "piano" :staff-short-name "pno"
    :lowest-written a0 :highest-written c8
    :largest-fast-leap 9
    :clefs (treble bass double-treble double-bass) :starting-clef treble
    :chords t :chord-function piano-chord-fun
    :microtones nil
    :midi-program 1))
;; We generally treat the piano as two instruments (LH, RH), generating
;; lines separately. So this is the same as the piano instrument but has
;; no staff-name and starts with bass clef. Use set-limits to change the
;; range of the two hands, as they're both set to be full piano range
;; here.
(piano-lh
 (:lowest-written a0 :highest-written c8
  ;; MDE Tue Aug 21 17:47:07 2012 -- to avoid the NIL ins name in CMN
  :staff-name "" :staff-short-name ""
  :largest-fast-leap 9
  :chords t :chord-function piano-chord-fun
  :clefs (treble bass double-treble double-bass) :starting-clef bass
  :microtones nil
  :midi-program 1))
;; MDE Wed Sep 12 12:51:00 2018 -- the 5 octave instrument...range varies
;; according to model of course
(harpsichord
 (:staff-name "harpsichord" :staff-short-name "hpd"
  :lowest-written f1 :highest-written f6
  :largest-fast-leap 9
  :clefs (treble bass) :starting-clef treble
  :chords t :chord-function piano-chord-fun
  :microtones nil
  :midi-program 7))
(harp
 (:staff-name "harp" :staff-short-name "hp"
  :lowest-written b0 :highest-written gs7
  :largest-fast-leap 12
  :clefs (treble bass) :starting-clef treble
  :chords t :chord-function piano-chord-fun
  :microtones nil :harmonics t
  :open-strings nil :nodes ((12 2))
  :midi-program 47))
;; MDE Fri Jul 24 20:18:16 2015 -- range typical for modern organ with
;; 61-key manuals and 32-key pedalboard
(organ
 (:lowest-written c2 :highest-written c7 :staff-name "organ"
  :staff-short-name "org" :largest-fast-leap 9 :chords t
  :chord-function piano-chord-fun :clefs (treble bass)
```

```
      :starting-clef treble :microtones nil :midi-program 20))
   (organ-pedals
    (:lowest-written c2 :highest-written g4 :staff-name "organ pedals"
     :staff-short-name "ped" :largest-fast-leap 6 :chords nil
     :clefs (bass) :starting-clef bass :microtones nil :midi-program 20))
   (percussion
    (:staff-name "percussion" :staff-short-name "perc" :staff-lines 5
     :lowest-written d4 :highest-written g5
     :starting-clef percussion
     :midi-program 1))
   (tambourine
    (:staff-name "tambourine" :staff-short-name "tmb" :staff-lines 1
     ;; MDE Fri Nov 13 16:05:46 2015 -- put on b4 not c4
     ;; :lowest-written c4 :highest-written c4
     :lowest-written b4 :highest-written b4
     :starting-clef percussion
     :midi-program 1))
   (guitar
    (:staff-name "guitar" :staff-short-name "gtr"
     :lowest-written e3 :highest-written b6 :transposition-semitones -12
     :largest-fast-leap 31
     :starting-clef treble
     :chords t :chord-function guitar-chord-selection-fun
     :microtones nil :harmonics t
     :open-strings (e5 b4 g4 d4 a3 e3) ; written!
     :open-string-marks (c1 c2 c3 c4 c5 c6)
     :midi-program 25))
   ;; MDE Wed Oct  9 12:21:22 2013
   (mandolin
    (:staff-name "mandolin" :staff-short-name "mln"
     :lowest-written g3 :highest-written c7
     :largest-fast-leap 25
     :starting-clef treble
     ;; mandolin has same tuning as the violin
     :open-strings (e5 a4 d4 g3)
     :open-string-marks (c1 c2 c3 c4) ; assuming guitar numbering strategy
     :chords t :chord-function violin-chord-selection-fun
     ;; there is no GM programme for mandolin so use either steel string
     ;; guitar (26) or banjo 106 perhaps
     :microtones nil :midi-program 26))
   (soprano ; voice
    (:staff-name "soprano" :staff-short-name "sop"
     :lowest-written c4 :highest-written c6
     :starting-clef treble :midi-program 54))
   (mezzo  ; voice
    (:staff-name "mezzo-soprano" :staff-short-name "mez"
```

```
      :lowest-written a3 :highest-written a5
      :starting-clef treble :midi-program 54))
   (alto ; voice
    (:staff-name "alto" :staff-short-name "alt"
     :lowest-written f3 :highest-written f5
     :starting-clef treble :midi-program 54))
   (countertenor ; voice
    (:staff-name "countertenor" :staff-short-name "ctr"
     :lowest-written e3 :highest-written e5
     :starting-clef treble-8vb :midi-program 54))
   (tenor ; voice
    (:staff-name "tenor" :staff-short-name "ten"
     :lowest-written c3 :highest-written c5
     :starting-clef treble-8vb :midi-program 54))
   (baritone ; voice
    (:staff-name "baritone" :staff-short-name "bar"
     :lowest-written a2 :highest-written a4
     :starting-clef bass :midi-program 54))
   (bass ; voice
    (:staff-name "bass" :staff-short-name "bas"
     :lowest-written e2 :highest-written e4
     :starting-clef bass :midi-program 54))
   (violin
    (:staff-name "violin" :staff-short-name "vln"
     :lowest-written g3 :highest-written c7
     :largest-fast-leap 13
     :starting-clef treble
     :chords t :chord-function violin-chord-selection-fun
     :microtones t :harmonics t
     :open-strings (e5 a4 d4 g3)
     :open-string-marks (i ii iii iv)
     :midi-program 41))
   (viola
    (:staff-name "viola" :staff-short-name "vla"
     :lowest-written c3 :highest-written f6
     :largest-fast-leap 13
     :clefs (alto treble) :starting-clef alto
     :chords t :chord-function viola-chord-selection-fun
     :microtones t :harmonics t
     :open-strings (a4 d4 g3 c3)
     :open-string-marks (i ii iii iv)
     :midi-program 42))
   (viola-d-amore
    (:staff-name "viola d'amore" :staff-short-name "vla d'am"
     :lowest-written a2 :highest-written f7
     :largest-fast-leap 13
```

```
   :clefs (alto treble) :starting-clef alto
   :chords t :chord-function nil
   :microtones t :harmonics t
   :midi-program 41))
(cello
 (:staff-name "cello" :staff-short-name "vc"
  ;; of course it can go higher but best not to algorithmically select
  ;; these
  :lowest-written c2 :highest-written a5
  :largest-fast-leap 12
  :clefs (bass tenor treble) :starting-clef bass
  :chords t :chord-function cello-chord-selection-fun
  :microtones t :harmonics t
  :open-strings (a3 d3 g2 c2)
  :open-string-marks (i ii iii iv)
  :midi-program 43))
(double-bass
 (:staff-name "double bass" :staff-short-name "db"
  :lowest-written e2 :highest-written g5 :transposition-semitones -12
  :prefers-notes low
  :largest-fast-leap 10
  :clefs (bass tenor treble) :starting-clef bass
  :chords nil
  :microtones t :harmonics t
  :open-strings (g3 d3 a2 e2) ; written!
  :open-string-marks (i ii iii iv)
  :midi-program 44))
(bass-guitar
 (:staff-name "bass guitar" :staff-short-name "b. gtr"
  :lowest-written e2 :highest-written g4 :transposition-semitones -12
  :prefers-notes low
  :largest-fast-leap 10
  :clefs (bass treble) :starting-clef bass
  :chords t
  :microtones nil :harmonics t
  :open-strings (g3 d3 a2 e2) ; written!
  :open-string-marks (c1 c2 c3 c4)
  :midi-program 33))
;; SAR Thu Apr 12 18:19:21 BST 2012: Added "computer" part for "silent"
;; parts in case the user would like to create rhythmically independent
;; computer parts.
;; MDE Jul 2012 -- changed to reflect more clefs (and removed percussion)
(computer
 (:staff-name "computer" :staff-short-name "comp"
  :lowest-sounding C-1 :highest-sounding bf8
  :clefs (treble bass double-treble double-bass)
```

```
      ;; MDE Thu Oct  8 17:13:35 2020, Heidhausen -- FX 4 (atmosphere)
      :midi-program 100
      ;; MDE Sat Aug  5 18:42:03 2017 --
      :microtones t
      :starting-clef treble)))))
```

12.2   instruments/cello-chord-selection-fun

[instruments] [Functions]

DESCRIPTION:

 Create a double-stop chord object using the core string-chord-selection-fun
 and a value of 'G2 for the open III string.

SYNOPSIS:

```
(let ((vc-III (make-pitch 'g2)))
  (defun cello-chord-selection-fun (curve-num index pitch-list pitch-seq
                                    instrument set)
```

12.3   instruments/chord-fun-aux

[instruments] [Functions]

DESCRIPTION:

 An auxiliary function that allows users to create moderately tailored chord
 functions by setting values for the number of notes in the current set to
 skip, the number of desired notes in the resulting chord, and the maximum
 span of the resulting chord in semitones.

 This function must be called within a call to the defun macro to create a
 new chord function, as demonstrated below.

ARGUMENTS:

 The first six arguments -- curve-num, index, pitch-list, pitch-seq,
 instrument, and set -- are inherited and not required to be directly
 accessed by the user.

 - skip: An integer that is the step by which the function skips through the
   subset of currently available pitches. A value of 2, for example, will
   instruct the method to build chords from every second pitch in that
   subset.

- An integer that is the number of pitches that should be in each resulting
  chord. If the list of pitches available to an instrument is too short to
  make a chord with x notes, a chord with fewer pitches may be made
  instead.
- An integer that is the largest interval in semitones allowed between the
  bottom and top notes of the chord. If a chord made with the specified
  number of notes surpasses this span, a chord with fewer pitches may be
  made instead.

EXAMPLE:

```
(defun new-chord-function (curve-num index pitch-list pitch-seq instrument set)
  (chord-fun-aux curve-num index pitch-list pitch-seq instrument set 4 3 14))
```

```
=> NEW-CHORD-FUNCTION
```

SYNOPSIS:

```
(defun chord-fun-aux (curve-num index pitch-list pitch-seq instrument set
                      skip num-notes max-span)
```

12.4   instruments/chord-fun1

[instruments] [Functions]

DESCRIPTION:

Generate three-note chords where possible, using every second pitch from
the list of pitches currently available to the given instrument from the
current set, and ensuring that none of the chords it makes span more than
an octave.

In all functions that take this format curve-num is the

SYNOPSIS:

```
(defun chord-fun1 (curve-num index pitch-list pitch-seq instrument set)
```

12.5   instruments/chord-fun2

[instruments] [Functions]

DESCRIPTION:

Generates 4-note chords where possible, using every third pitch from the
list of pitches currently available to the given instrument from the
current set, with (almost) no limit on the total span of the chord.

SYNOPSIS:

(defun chord-fun2 (curve-num index pitch-list pitch-seq instrument set)

12.6   instruments/get-standard-ins

[instruments] [Functions]

DATE:

August 29th 2018, Heidhausen

DESCRIPTION:

Convenience function for get-data from the standard instrument palette

ARGUMENTS:

- the ID of the instrument (symbol)

OPTIONAL ARGUMENTS:

- either NIL (no error/warning) or a function (object) to call if the
  instrument isn't found. Default = #'error

RETURN VALUE:

the instrument object

SYNOPSIS:

(defun get-standard-ins (ins &optional (on-fail #'error))

12.7   instruments/guitar-chord-selection-fun

[instruments] [Functions]

DESCRIPTION:

Create chord objects with differing numbers of pitches, drawing the pitches
from set-palette object subsets with the ID 'guitar.

This function was written for the composition "Cheat Sheet", in which the
pitch sets were defined explicitly such that all of the pitches available
to the guitar at any moment were playable as a guitar chord. As such, this
function always assumes that the pitch-list it is drawing from contains
pitches that are already playable as a guitar chord. It also adds the
fingering as mark above each chord when outputting to CMN, which may or may
not be desirable.

SYNOPSIS:

```
(let ((last-chord '()))
  (defun guitar-chord-selection-fun (curve-num index pitch-list pitch-seq
                                     instrument set)
```

12.8   instruments/natural-harmonic

[instruments] [Functions]

DATE:

December 24th 2013

DESCRIPTION:

Determine whether a pitch can be played as a natural harmonic on a (string)
instrument (with the guitar as default).

MDE Tue Jan 8 18:22:09 2019 -- This function is retained for historical
reasons but has been superseded by the instrument method natural-harmonic?
along with the associated instrument slots open-strings, open-string-marks,
and nodes (which altogether are more flexible and efficient than this
function alone)

ARGUMENTS:

- the pitch (symbol or pitch object)

OPTIONAL ARGUMENTS:

keyword arguments:
- :tuning. a list of the fundamentals of the open strings, as pitch objects
  or symbols. These should descend from the highest string. Default:

     guitar tuning.
   - :highest-partial. Integer. What we consider the highest harmonic possible
     (counting the fundamental as 1).
   - :tolerance. The deviation in cents that we can accept for the frequency
     comparison. Default = 10.
   - :debug. Print data for debugging/testing purposes. Default = NIL.

RETURN VALUE:

 The string number and partial number as a list if possible as a harmonic,
 or NIL if not.

EXAMPLE:

```
(NATURAL-HARMONIC 'b5) ; octave harmonic of B string
=> (2 2)
SC> (NATURAL-HARMONIC 'b6) ; octave + 5th of high E string
=> (1 3)
```

SYNOPSIS:

```
(defun natural-harmonic (pitch &key (tuning '(e5 b4 g4 d4 a3 e3))
                        (highest-partial 6) (tolerance 15) debug)
```

12.9   instruments/piano-chord-fun

[instruments] [Functions]

DESCRIPTION:

 Generate four-note chords, where possible, from consecutive notes in the
 current set, with the number enclosed in parentheses in the pitch-seq being
 the top note of that chord, where possible.

SYNOPSIS:

```
(defun piano-chord-fun (curve-num index pitch-list pitch-seq instrument set)
```

12.10   instruments/piano-chord-fun3

[instruments] [Functions]

DATE:

 August 2nd 2019, Heidhausen

DESCRIPTION:

 Almost exactly the same as piano-chord-fun but generally get just three
 notes rather than four. If we want to force a maximum of three notes, it
 will need to be done with subseq or similar. I find this 'usually three
 notes' preferable though as some four-note chords are returned, for variety.

SYNOPSIS:

(defun piano-chord-fun3 (curve-num index pitch-list pitch-seq instrument set)

12.11   instruments/play-all

[instruments] [Functions]

DESCRIPTION:

 A very simple chord function to simply play all the notes in the set, no
 matter which instrument should play them. Useful perhaps for computer
 voices. In any case there's no checking of instrument range as we ignore
 the pitch-list here.

SYNOPSIS:

(defun play-all (curve-num index pitch-list pitch-seq instrument set)

12.12   instruments/set-standard-range

[instruments] [Functions]

DATE:

 September 24th 2018, Heidhausen

DESCRIPTION:

 Set the lowest and highest pitches of an instrument in the standard
 instrument palette. Use this function to correctly set the highest sounding
 and written slots of transposing and non-transposing instruments alike.

ARGUMENTS:

 - the instrument id (symbol)
 - the lowest pitch (symbol or pitch object). This can also be NIL, whereupon
   no change will be made to the current slot's value.
 - the highest pitch (symbol or pitch object). Can also be NIL.

OPTIONAL ARGUMENTS:

 – T or NIL to indicated whether the pitches given are to be interpreted as
   written or sounding pitches. In either case, both written and sounding
   pitches will be set for the instrument. Default = NIL = written.

RETURN VALUE:

 The instrument object for which the changes were made.

SYNOPSIS:

(defun set-standard-range (ins low high &optional sounding)

12.13   instruments/string-chord-selection-fun

[instruments] [Functions]

DESCRIPTION:

 This is the core function for creating instances of double-stop chords for
 strings, ensuring that the highest note of the double-stop is not lower
 than the open III string. The pitch of the open III string is passed as an
 argument in the chord-selection functions for the individual stringed
 instruments.

 This function uses the best-string-diad function. If no double-stops
 instances can be created using best-string-diad, two-note chords will be
 created using the default-chord-function. If neither of these are possible,
 a chord of a single pitch will be returned instead.

SYNOPSIS:

(defun string-chord-selection-fun (curve-num index pitch-list pitch-seq
                                   instrument set string-III)

12.14   instruments/viola-chord-selection-fun

[instruments] [Functions]

DESCRIPTION:

 Create a double-stop chord object using the core string-chord-selection-fun
 and a value of 'G3 for the open III string.

SYNOPSIS:

```
(let ((vla-III (make-pitch 'g3)))
  (defun viola-chord-selection-fun (curve-num index pitch-list pitch-seq
                                    instrument set)
```

12.15   instruments/violin-chord-selection-fun

[instruments] [Functions]

DESCRIPTION:

```
 Create a double-stop chord object using the core string-chord-selection-fun
 and a value of 'D4 for the open III string.
```

SYNOPSIS:

```
(let ((vln-III (make-pitch 'd4)))
  (defun violin-chord-selection-fun (curve-num index pitch-list pitch-seq
                                     instrument set)
```

13   sc/lilypond

[Modules]

13.1   lilypond/lp-get-mark

[lilypond] [Functions]

DESCRIPTION:

```
 lp-get-mark:
 Translation function for LilyPond marks (dynamics, accents, etc.).  Not
 generally called by the user but the list of symbols that can be used will
 be useful.  If <silent> then non-existing marks will not produce
 warnings/errors (but we'll return nil).
 MDE Thu Sep 15 10:50:47 2016 -- default to the
 config setting
```

SYNOPSIS:

```
        (a "-> ")                        ; accent
        (lhp "-+ ")
```

```
;; see p229 of lilypond.pdf: need to define this command in file
;; this is done for us in lilypond.ly, which will be included if we
;; call write-lp-data-for-all with :use-custom-markup T
(bartok "^\\snapPizzicato ")
(pizz "^\"pizz.\" ")
(ord "^\"ord.\" ")
(pizzp "^\"(pizz.)\" ")
(clb "^\"clb\"")
(cl "^\"cl\" ")
(col-legno "^\"col legno\" ")
(clt "^\"clt\" ")
(arco "^\"arco\" ")
(batt "^\"batt.\" ")
(spe "^\"spe\" ")
(sp "^\"sul pont.\" ")
(st "^\"sul tasto\" ")
(mv "^\"molto vib.\" ")
(sv "^\"senza vib.\" ")
(poco-crini "^\"poco crini\" ")
(s "-. ")
(nail (no-lp-mark 'nail))
(stopped (no-lp-mark 'stopped))
(as "->-. ")
(at "->-- ")
(ts "-_ ")
(te "-- ")
;; so unmeasured is implicit
(t3 (format nil ":~a " (* 32 (expt 2 num-flags))))
(flag "\\flageolet ")
(niente "^\markup { niente } ")
(pppp "\\pppp ")
(ppp "\\ppp ")
(pp "\\pp ")
(p "\\p ")
(mp "\\mp ")
(mf "\\mf ")
(f "\\f ")
(ff "\\ff ")
(fff "\\fff ")
(ffff "\\ffff ")
;; MDE Sat Aug 11 15:51:16 2012 -- dynamics in parentheses
(ffff-p "\\parenFFFF ")
(fff-p "\\parenFFF ")
(ff-p "\\parenFF ")
(f-p "\\parenF ")
(mf-p "\\parenMF ")
```

```
(mp-p "\\parenMP ")
(p-p "\\parenP ")
(pp-p "\\parenPP ")
(ppp-p "\\parenPPP ")
(pppp-p "\\parenPPPP ")
(sfz "\\sfz ")
(downbow "\\downbow ")
(upbow "\\upbow ")
(open "\\open ")
(I "^\\markup { \\teeny \"I\" } ")
(II "^\\markup { \\teeny \"II\" } ")
(III "^\\markup { \\teeny \"III\" } ")
(IV "^\\markup { \\teeny \"IV\" } ")
;; MDE Thu Dec 26 14:14:34 2013 -- guitar string numbers
(c1 "\\1 ")
(c2 "\\2 ")
(c3 "\\3 ")
(c4 "\\4 ")
(c5 "\\5 ")
(c6 "\\6 ")
(beg-sl "( ")
(end-sl  ") ")
;; MDE Fri Apr  6 21:57:59 2012 -- apparently LP can't have nested
;; slurs but it does have phrase marks:
(beg-phrase "\\( ")
(end-phrase "\\) ")
;; there's no start gliss / end gliss in lilypond
(beg-gliss "\\glissando ")
(end-gliss "")
;; 13.4.11
(beg-8va "\\ottava #1 ")
(end-8va "\\ottava #0 ")
(beg-8vb "\\ottava #-1 ")
(end-8vb "\\ottava #0 ")
;; MDE Mon Jan 18 11:43:43 2016
(beg-15ma "\\ottava #2 ")
(end-15ma "\\ottava #0 ")
(beg-15mb "\\ottava #-2 ")
(end-15mb "\\ottava #0 ")
;; NB note heads used to be added via (add-mark-before) but can now
;; be added by add-mark and friends (16.6.20) so if adding new heads
;; , make sure to add the mark symbol to the move-elements call in
;; event::separate-marks-before
;;
;; (circled-x "\\once \\override NoteHead #'style = #'xcircle ")
(circled-x "\\tweak #'style #'xcircle ")
```

```
;; (x-head "\\once \\override NoteHead #'style = #'cross ")
(x-head "\\xNote ")
;; (triangle "\\once \\override NoteHead #'style = #'triangle ")
(triangle "\\tweak #'style #'triangle ")
;; (triangle-up "\\once \\override NoteHead #'style = #'do ")
(triangle-up "\\tweak #'style #'do ")
(airy-head (no-lp-mark 'airy-head))
;; this has to be added to the event _before_ the one which needs to
;; start with these noteheads.
(improvOn "\\improvisationOn ")
(improvOff "\\improvisationOff ")
;; MDE Sat Nov  9 20:21:19 2013 -- in CMN it's :breath-in: a
;; triangle on its side (pointing left)
;; (wedge "\\once \\override NoteHead #'style = #'fa ")
(wedge "\\tweak #'style #'fa ")
;; (square "\\once \\override NoteHead #'style = #'la ")
(square "\\tweak #'style #'la ")
;; (mensural "\\once \\override NoteHead #'style = #'slash ")
;;(flag-head "\\once \\override NoteHead #'style = #'harmonic-mixed
;;")
;; MDE Mon Apr 30 20:46:06 2012 -- see event::get-lp-data for how
;; this is handled
(flag-head "\\harmonic ") ; diamond
;; MDE Mon Apr 30 20:46:31 2012 -- flag-heads by default don't
;; display dots so we need to add-mark-before to get these to
;; display or turn them off again
(flag-dots-on "\\set harmonicDots = ##t ")
(flag-dots-off "\\set harmonicDots = ##f ")
;; circle head but stem extends through it like a vertical slash
(none (no-lp-mark 'none))
(trill-f (no-lp-mark 'trill-f))
(trill-n (no-lp-mark 'trill-n))
(trill-s (no-lp-mark 'trill-s))
(beg-trill-a "\\pitchedTrill ") ; must be before note
;; we'll also need e.g. (trill-note g5) to give the note in ()
(end-trill-a "\\stopTrillSpan ") ; after note
;; (no-lp-mark 'square))
(slash (no-lp-mark 'slash))
;; MDE Sat Dec 28 11:37:22 2013 -- up and down arrows on arpeggio
;; lines will need more complex treatment (need a note-before mark
;; :/ )
(arp "\\arpeggio ")
(arrow-up (no-lp-mark 'arrow-up))
(arrow-down (no-lp-mark 'arrow-down))
(cresc-beg "\\< ")
(cresc-end "\\! ")
```

```
(dim-beg "\\> ")
(dim-end "\\! ")
(<< "<< ")
(>> ">> ")
;; NB this override has to come exactly before the note/dynamic it
;; applies to. MDE Tue Jun 16 23:38:57 2020, Heidhausen -- again this
;; override should be OK (rather than tweak) as it's below the staff
;; rather than a property of a note (in a chord)
(hairpin0 "\\once \\override Hairpin #'circled-tip = ##t ")
;; (dim0-beg "\\once \\override Hairpin #'circled-tip = ##t \\> ")
(pause "\\fermata ")
(long-pause "\\longfermata ")
(short-pause
 "^\\markup { \\musicglyph #\"scripts.ushortfermata\" } ")
;; MDE Thu Apr  5 16:17:11 2012 -- these need the graphics files in
;; lilypond-graphics.zip to be in the same directory as the
;; generated lilypond files
(aeolian-light "^\\aeolianLight ")
(aeolian-dark "^\\aeolianDark ")
;; this one uses the graphic for close bracket
(bracket-end "^\\bracketEnd ")
(mphonic "^\\mphonic ")
(mphonic-arr "^\\mphonicArr ")
(mphonic-cons "^\\mphonicCons ")
(mphonic-diss "^\\mphonicDiss ")
(mphonic-cluster "^\\mphonicCluster ")
(sing "^\\sing ")
(high-sine "^\\high-sine ")
(noise "^\\noise ")
(focus "^\\focus ")
(balance "^\\balance ")
(alternate "^\\alternate ")
(sing-arr "^\\singArr ")
(arrow-up-down "^\\arrowUpDown ")
;; end lilypond-graphics.zip files
;; these must have been set up with the event::add-arrow method
(start-arrow "\\startTextSpan ")
(end-arrow "\\stopTextSpan ")
(harm "^\\flageolet ")
;; 2.3.11
;; write sost. pedal as text (usually held for long time so brackets
;; not a good idea)
;; MDE Wed May 25 12:40:57 2016 -- update: do use the LP
;; sost. brackets after all
(sost "\\sostenutoOn ")
(sost-up "\\sostenutoOff ")
```

```
(sost^ "\\sostenutoOff\\sostenutoOn ")
(ped "\\sustainOn ")
(ped^ "\\sustainOff\\sustainOn ")
(ped-up "\\sustainOff ")
(uc "\\unaCorda ")
(tc "\\treCorde ")
```

14   sc/music-xml

[Modules]

14.1   music-xml/xml-get-mark

[music-xml] [Functions]

DESCRIPTION:

```
xml-get-mark:
Translation function for music-xml marks (dynamics, accents, etc.). Not
generally called by the user but the list of symbols that can be used will
be useful. If <silent> then non-existing marks will not produce
warnings/errors (but we'll return nil).
default to the config setting
```

SYNOPSIS:

```
;; if returns a string then xml-articulation will be called,
;; otherwise (dir ...) (tech ...)
;; also possible: (art "accent" "below") or a list of
;; strings, in which case there'll be 2+ xml-articulation
(a "accent")        ; accent
(s "staccato")
(as '("accent" "staccato"))
(at '("accent" "tenuto"))
(ts "detached-legato")
(te "tenuto")
;; tech means xml-technical will be called; see bottom of
;; this function
(lhp '(tech "stopped" a))
(bartok '(tech "snap-pizzicato" a))
;; todo: could also add the <sound> tag here to change
;; midi-programme (see MozartTrio.xml for e.g.)
;; dir means xml-direction will be called
(pizz '(wds "pizz." a))
```

```
(ord '(wds "ord."))
(pizzp '(wds "(pizz.)"))
(clb '(wds "clb"))
(cl '(wds "cl"))
(col-legno '(wds "col legno"))
(clt '(wds "clt"))
(arco '(wds "arco"))
(batt '(wds "batt"))
(spe '(wds "spe"))
(sp '(wds "sul pont."))
(st '(wds "sul tasto"))
(mv '(wds "molto vib"))
(sv '(wds "senza vib"))
(poco-crini '(wds "poco crini"))
(nail '(tech "fingernails" a))
(stopped (no-xml-mark 'stopped))
;; so unmeasured tremolo is implicit
(t3 '(orn "tremolo" nil 3))
(flag '(tech "harmonic" a))
;; MDE Fri Dec  7 18:11:29 2018 -- dorico still doesn't handle
;; the little circles to start/end a hairpin
(niente '(wds "niente"))
(pppp  '(dyn "pppp"))
(ppp  '(dyn "ppp"))
(pp  '(dyn "pp"))
(p  '(dyn "p"))
(mp  '(dyn "mp"))
(mf  '(dyn "mf"))
(f  '(dyn "f"))
(ff  '(dyn "ff"))
(fff  '(dyn "fff"))
(ffff  '(dyn "ffff"))
;; todo: find out how to put dynamics in parentheses; for now
;; just put in the dynamic without
(pppp-p '(dyn "pppp"))
(ppp-p '(dyn "ppp"))
(pp-p '(dyn "pp"))
(p-p '(dyn "p"))
(mp-p '(dyn "mp"))
(mf-p '(dyn "mf"))
(f-p '(dyn "f"))
(ff-p '(dyn "ff"))
(fff-p '(dyn "fff"))
(ffff-p '(dyn "ffff"))
(sfz '(dyn "sfz"))
(downbow '(tech "down-bow" a))
```

```
(upbow '(tech "up-bow" a))
(open '(tech "fingering" a 0))
(I '(wds "I"))
(II '(wds "II"))
(III '(wds "III"))
(IV '(wds "IV"))
(V '(wds "V"))
(VI '(wds "VI"))
(VII '(wds "VII"))
(VIII '(wds "VIII"))
;; guitar string numbers; todo: find a way of circling them
(c1 "1")
(c2 "2")
(c3 "3")
(c4 "4")
(c5 "5")
(c6 "6")
;; music xml doesn't seem to have separate slur and phrase
;; marks so we'll simply use the slur tag with number=1 for
;; slurs and number=2 for phrases
(beg-sl '(not "slur" "number=\"1\" type=\"start\""))
(end-sl '(not "slur" "number=\"1\" type=\"stop\""))
(beg-phrase '(not "slur" "number=\"2\" type=\"start\""))
(end-phrase '(not "slur" "number=\"2\" type=\"stop\""))
(beg-gliss '(not "glissando" "number=\"1\" type=\"start\" line-type=\"solid\""))
(end-gliss '(not "glissando" "number=\"1\" type=\"stop\" line-type=\"solid\""))
(beg-8va '(dir "octave-shift"
            ;; it really is down, probably because what's
            ;; notated is an 8ve lower than sounding
            "dash-length=\"7.5\" size=\"8\" space-length=\"7.5\" type=\"down\""))
(end-8va '(dir "octave-shift" "size=\"8\" type=\"stop\""))
(beg-8vb '(dir "octave-shift"
            "dash-length=\"7.5\" size=\"-8\" space-length=\"7.5\" type=\"up\""))
(end-8vb '(dir "octave-shift" "size=\"-8\" type=\"stop\""))
(beg-15ma '(dir "octave-shift"
             "dash-length=\"7.5\" size=\"15\" space-length=\"7.5\" type=\"down\""))
(end-15ma '(dir "octave-shift" "size=\"15\" type=\"stop\""))
(beg-15mb '(dir "octave-shift"
             "dash-length=\"7.5\" size=\"-15\" space-length=\"7.5\" type=\"up\""))
(end-15mb '(dir "octave-shift" "size=\"-15\" type=\"stop\""))
;; note heads
(circled-x '(hd "circle-x"))
(x-head '(hd "x"))
(triangle '(hd "triangle"))
(triangle-up '(hd "inverted triangle"))
(airy-head (no-xml-mark 'airy-head))
```

```
                      (none '(hd "none"))
                      ;; the On and Off versions were for lilypond; here we only
                      ;; need the On version (or preferably, just improv but for
                      ;; compatibility...)
                      (improvOn '(hd "slash"))
                      (improvOff nil)
                      (wedge '(hd "fa"))
                      (square '(hd "la"))
                      ;; todo: look at xml doc for harmonic and sort out a way of
                      ;; representing artificial harms in xml
                      (flag-head '(hd "diamond"))
                      (flag-dots-on (no-xml-mark 'flag-dots-on))
                      (flag-dots-off (no-xml-mark 'flag-dots-off))
                      (trill-f (no-xml-mark 'trill-f))
                      (trill-n (no-xml-mark 'trill-n))
                      (trill-s (no-xml-mark 'trill-s))
                      (beg-trill-a (xml-beg-trill stream))
                      (end-trill-a (xml-end-trill stream))
                      (slash (no-xml-mark 'slash))
                      (arp '(not "arpeggiate" "default-x=\"-12\"  number=\"1\""))
                      (arrow-up (no-xml-mark 'arrow-up))
                      (arrow-down (no-xml-mark 'arrow-down))
                      (cresc-beg '(dir "wedge" nil
                                  "spread=\"0\" type=\"crescendo\"" b))
                      (cresc-end '(dir "wedge" nil
                                  "spread=\"15\" type=\"stop\"" b))
                      (dim-beg '(dir "wedge" nil
                                "spread=\"15\" type=\"diminuendo\"" b))
                      (dim-end '(dir "wedge" nil
                                "spread=\"0\" type=\"stop\"" b))
                      (<< (no-xml-mark '<<))
                      (>> (no-xml-mark '>>))
                      ;; we could faff around with lexical variables and manipulate
                      ;; the niente property of "wedge" but as this would be
                      ;; post-gen anyway we can do it in the notation software
                      (hairpin0 (no-xml-mark 'hairpin0))
                      (pause (xml-pause stream))
                      ;; seems there is a fermata-shape element but damned if I can
                      ;; find out where it could go and be interpreted correctly.
                      ;; MDE Mon Mar 27 13:09:50 2017 -- see xml-pause above
                      (long-pause (xml-pause stream)); "lunga"))
                      (short-pause (xml-pause stream)); "breva"))
                      (aeolian-light (no-xml-mark 'aeolianLight))
                      (aeolian-dark (no-xml-mark 'aeolianDark))
                      ;; this one uses the graphic for close bracket
                      (bracket-end (no-xml-mark 'bracketEnd))
```

```
(mphonic (no-xml-mark 'mphonic))
(mphonic-arr (no-xml-mark 'mphoniArr))
(mphonic-cons (no-xml-mark 'mphonicCons))
(mphonic-diss (no-xml-mark 'mphonicDiss))
(mphonic-cluster (no-xml-mark 'mphonicCluster))
(sing (no-xml-mark 'sing))
(high-sine (no-xml-mark 'high-sine))
(noise (no-xml-mark 'noise))
(focus (no-xml-mark 'focus))
(balance (no-xml-mark 'balance))
(alternate (no-xml-mark 'alternate))
(sing-arr (no-xml-mark 'singArr))
(arrow-up-down (no-xml-mark 'arrowUpDown))
;; no need to fiddle with these: can do in notation software,
;; no?
(start-arrow (no-xml-mark 'start-arrow))
(end-arrow (no-xml-mark 'end-arrow))
(harm '(tech "harmonic" a "<natural />"))
(sost '(wds "sost."))
(sost-up '(wds "sost.*"))
(sost^ (no-xml-mark 'sost^)) ;"\\sostenutoOff\\sostenutoOn ")
(ped '(dir "pedal" nil "line=\"yes\" type=\"start\""))
(ped^ '(dir "pedal" nil "line=\"yes\" type=\"change\""))
(ped-up '(dir "pedal" nil "line=\"yes\" type=\"stop\""))
(uc '(wds "una corda" nil b))
(tc '(wds "tre corde" nil b))
```

15   sc/package

[Modules]

16   sc/permutations

[Modules]

NAME:

```
 permutations

 File:            permutations.lsp

 Class Hierarchy:  none, no classes defined.
```

```
Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Various permutation functions.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     10th November 2002

$$ Last modified:  10:26:32 Sat Jun  6 2020 CEST

SVN ID: $Id$
```

16.1   permutations/permutations

[permutations] [Functions]

DESCRIPTION:

Systematically produce a list of all possible permutations of a set of
consecutive integers beginning with zero. The function's only argument,
<level>, is an integer that determines how many consecutive integers from 0
are to be used for the process.

This is a more efficient permutation algorithm, but the results will always
be in a certain order, with the same number at the end until that
permutation is exhausted, then the number below that etc.

ARGUMENTS:

An integer that indicates how many consecutive integers from 0 are to be
used for the process.

OPTIONAL ARGUMENTS:

- allow-file: whether to allow the results to be written to a file rather
than returned to the interpreter. This will only happen if the length of
the list is > 8. The advantage is that you can then read in the file rather
than slowly regenerate the results if you need them again.

RETURN VALUE:

A list of sequences (lists), each of which is a permutation of the
original.

EXAMPLE:

```
;; Produce a list consisting of all permutations that can be made of 4
;; consecutive integers starting with 0 (i.e., (0 1 2 3))
(permutations 4)

=>
((0 1 2 3) (1 0 2 3) (0 2 1 3) (2 0 1 3) (1 2 0 3) (2 1 0 3) (0 1 3 2)
 (1 0 3 2) (0 3 1 2) (3 0 1 2) (1 3 0 2) (3 1 0 2) (0 2 3 1) (2 0 3 1)
 (0 3 2 1) (3 0 2 1) (2 3 0 1) (3 2 0 1) (1 2 3 0) (2 1 3 0) (1 3 2 0)
 (3 1 2 0) (2 3 1 0) (3 2 1 0))
```

SYNOPSIS:

```
(defun permutations (level &optional (allow-file t))
```

16.1.1   permutations/avoid-common-elements

[permutations] [Functions]

DATE:

 December 6th 2019, Heidhausen

DESCRIPTION:

 Taking a list of sublists, move the sublists around to try and avoid
 consecutive sublists having common elements. Note that even if it's not
 possible to completely solve this problem, a re-ordered list will be
 returned along with a warning detailing where things were no longer
 possible. No further, more complex attempts are made at that point. NB In
 that case shuffling before calling this function might help

ARGUMENTS:

 - the list of sublists (of any element type)

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :test. The function to be passed to (intersection) for testing equivalence
   amongst list elements. Default = #'eq
 - :accept. The number of common elements to accept amongst sublists. Default
   = 0
 - :warn. Issue a warning if we can't complete? Default = T

- :all-on-fail. Return the unprocessed (failing) elements if we can't
  complete? If NIL then we'll return less sublists than given but there will
  be no common elements, at least. Default = T.

RETURN VALUE:

 a list: the first argument with re-ordered sublists.

EXAMPLE:

```
(avoid-common-elements
 '((1 2 3) (4 5 6) (5 4 6) (4 6 1) (2 3 4) (2 3 1) (1 3 2)))
-->
WARNING: permutations::avoid-common-elements: did 4 but can't complete.
Current: (1 3 2)
Rest: ((4 6 1) (2 3 4))
((1 2 3) (4 5 6) (2 3 1) (5 4 6) (1 3 2) (4 6 1) (2 3 4))

(avoid-common-elements
 '((1 2 3) (4 5 6) (5 4 6) (3 2 1) (2 3 1) (6 5 4) (3 1 2)))
-->
((1 2 3) (4 5 6) (3 2 1) (5 4 6) (2 3 1) (6 5 4) (3 1 2))
```

SYNOPSIS:

```
(defun avoid-common-elements (lists &key (test #'eq) (accept 0) (warn t)
                                    (all-on-fail t))
```

16.1.2   permutations/inefficient-permutations

[permutations] [Functions]

DESCRIPTION:

 Return a shuffled, non-systematic list of all possible permutations of a
 set of consecutive integers beginning with zero.

 The function's first argument, <level>, is an integer that determines how
 many consecutive integers from 0 are to be used for the process. An
 optional keyword argument <max> allows the user to specify the maximum
 number of permutations to return.

 This function differs from the "permutations" function in that it's result
 is not ordered systematically.

The function simply returns a list of <max> permutations of the numbers
less than <level>; it does not permutate a given list.

The function is inefficient in so far as it simply shuffles the numbers and
so always has to check whether the new list already contains the shuffled
before storing it.

The order of the permutations returned will always be the same unless <fix>
is set to NIL.

Keyword argument <skip> allows the user to skip a number of permutations,
which is only sensible if :fix is set to T.

ARGUMENTS:

An integer that indicates how many consecutive integers from 0 are to be
used for the process.

OPTIONAL ARGUMENTS:

keyword arguments:
- :max. An integer that indicates the maximum number of permutations to be
  returned.
- :skip. An integer that indicates a number of permutations to skip.
- :fix. T or NIL to indicate whether the given sequence should always be
  shuffled with the same (fixed) random seed (thus always producing the
  same result). T = fixed seed. Default = T.
- :if-not-enough. A function object (or NIL) to call when :max was
  requested but we can't return that many results. Default = #'error.

RETURN VALUE:

A list.

EXAMPLE:

```
;; Creating a shuffled, non-systematic list of all permutations of consecutive
;; integers 0 to 4
(inefficient-permutations 4)

=> ((2 3 0 1) (3 1 2 0) (2 0 3 1) (1 0 2 3) (1 2 3 0) (0 2 3 1) (2 1 0 3)
    (0 1 2 3) (2 3 1 0) (1 2 0 3) (3 0 1 2) (3 1 0 2) (1 3 2 0) (1 0 3 2)
    (2 0 1 3) (3 2 1 0) (2 1 3 0) (3 2 0 1) (1 3 0 2) (0 2 1 3) (3 0 2 1)
    (0 1 3 2) (0 3 2 1) (0 3 1 2))
```

```
;; Using 0 to 4 again, but limiting the number of results returned to a maximum
;; of 7
(inefficient-permutations 4 :max 7)

=> ((2 3 0 1) (3 1 2 0) (2 0 3 1) (1 0 2 3) (1 2 3 0) (0 2 3 1) (2 1 0 3))

;; The same call will return the same "random" results each time by default
(loop repeat 4 do (print (inefficient-permutations 3 :max 5)))

=>
((2 0 1) (2 1 0) (0 2 1) (1 0 2) (1 2 0))
((2 0 1) (2 1 0) (0 2 1) (1 0 2) (1 2 0))
((2 0 1) (2 1 0) (0 2 1) (1 0 2) (1 2 0))
((2 0 1) (2 1 0) (0 2 1) (1 0 2) (1 2 0))

;; Setting the :fix argument to NIL will result in different returns
(loop repeat 4 do (print (inefficient-permutations 3 :max 5 :fix nil)))

=>
((1 0 2) (0 1 2) (1 2 0) (2 1 0) (0 2 1))
((1 2 0) (2 0 1) (2 1 0) (1 0 2) (0 1 2))
((0 1 2) (1 0 2) (2 0 1) (1 2 0) (2 1 0))
((0 2 1) (1 2 0) (0 1 2) (2 0 1) (1 0 2))
```

SYNOPSIS:

```
(defun inefficient-permutations (level &key (max nil) (skip 0) (fix t)
                                            (if-not-enough #'error))
```

16.1.3   permutations/inefficiently-permutate

[permutations] [Functions]

DESCRIPTION:

Return a shuffled, non-systematically ordered list of all possible
permutations of an original list of elements of any type. An optional
keyword argument <max> allows the user to specify the maximum number of
permutations to return.

As opposed to the function "permutate", inefficiently-permutate returns the
elements of the specified <list> as a flat list, unless the keyword
argument <sublists> is set to T, whereupon the function returns the result
as a list of lists, each one being a permutation of <list>.

The function is inefficient in so far as it simply shuffles the numbers and
so always has to check whether the new list already contains the shuffled
sublist before storing it.

The order of the permutations returned will always be the same unless <fix>
is set to NIL.

ARGUMENTS:

- A list.

OPTIONAL ARGUMENTS:

keyword arguments:
- :max. An integer that indicates the maximum number of permutations to be
  returned.
- :skip. An integer that indicates a number of permutations to skip.
- :fix. T or NIL to indicate whether the given sequence should always be
  shuffled with the same (fixed) random seed (thus always producing the
  same result). T = fixed seed. Default = T.
- :sublists. T or NIL to indicate whether the returned result should be
  flattened into a one-dimensional list or should be left as a list of
  lists. T = leave as list of lists. Default = NIL.
- :clone. T or NIL to indicate whether objects in the list should be cloned
  as they are permutated (so that they are unique objects rather than
  shared data space). Useful perhaps if e.g. you're cloning chords which
  will then have their own marks. etc. If T then the list must contain
  slippery-chicken named-objects or types subclassed from them (as is every
  slippery-chicken class).
- :if-not-enough. A function object (or NIL) to call when :max was
  requested but we can't return that many results. Default = #'error.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; By default the function returns a flattened list of all possible
;; permutations in a shuffled (random) order
(inefficiently-permutate '(a b c))

=> (C A B C B A A C B B A C B C A A B C)

;; The length of the list returned can be potentially shortened using the :max
```

```
;; keyword argument. Note here that the value given here refers to the number
;; of permutations before the list is flattened, not to the number of
;; individual items in the flattened list.
(inefficiently-permutate '(a b c) :max 3)

=> (C A B C B A A C B)

;; By default the function is set to using a fixed random seed, causing it to
;; return the same result each time
(loop repeat 4 do (print (inefficiently-permutate '(a b c))))

=>
(C A B C B A A C B B A C B C A A B C)
(C A B C B A A C B B A C B C A A B C)
(C A B C B A A C B B A C B C A A B C)
(C A B C B A A C B B A C B C A A B C)

;; Setting the :fix keyword argument to NIL allows the function to produce
;; different output each time
(loop repeat 4 do (print (inefficiently-permutate '(a b c) :fix nil)))

=>
(B A C A C B B C A A B C C B A C A B)
(A C B B A C C B A C A B B C A A B C)
(A C B B A C B C A A B C C A B C B A)
(B A C A B C C A B C B A B C A A C B)

;; Setting the :sublists keyword argument to T causes the function to return a
;; list of lists instead
(inefficiently-permutate '(a b c) :sublists t)

=> ((C A B) (C B A) (A C B) (B A C) (B C A) (A B C))
```

SYNOPSIS:

```
(defun inefficiently-permutate (list &key (max nil) (skip 0) (fix t)
                                 clone (sublists nil) (if-not-enough #'error))
```

16.1.4   permutations/move-repeats

[permutations] [Functions]

DESCRIPTION:

 Move, when possible, any elements within a given list that are repeated

consecutively.

When two consecutive elements repeat, such as the c in '(a b c c b a),
the function moves the repeated element to the next place in the given
list that won't produce a repetition. When no such place can be found in
the remainder of the list, the offending element is moved to the end of the
given list and a warning is printed.

This function can be applied to simple lists and lists with sublists.
However, due to this function being designed for--but not limited to--use
with the results of permutations, if the list has sublists, then instead of
repeating sublists being moved, the last element of a sublist is checked
for repetition with the first element of the next sublist. See the first
example below.

NB: This function only moves elements further along the list; it won't place
   them earlier than their original position. Thus:

   (move-repeats '(3 3 1))

   will return (3 1 3), while

   (move-repeats '(1 3 3))

   will leave the list untouched and print a warning.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

 - A function that serves as the comparison test. Default = #'eq.

RETURN VALUE:

 A list.

EXAMPLE:

```
;;; Used with a list of lists.  Note that the repeating C, end of sublist 1,
;;; beginning of sublist 2, is moved, not the whole repeating sublist (c a b).
(move-repeats '((a b c) (c a b) (c a b) (d e f) (a b c) (g h i)))

=> ((A B C) (D E F) (C A B) (C A B) (A B C) (G H I))
```

```
;;; Works with simple lists too:
(move-repeats '(1 2 3 3 4 5 6 7 8 8 9 10))

=> (1 2 3 4 3 5 6 7 8 9 8 10)

;; Moves the offending element to the end of the list and prints a warning when
;; no solution can be found
(move-repeats '((a b c d) (d c b a) (b c a d) (c a b d)))

=> ((A B C D) (B C A D) (C A B D) (D C B A))
WARNING:
   move-repeats: can't find non-repeating place!
   present element: (D C B A), elements left: 1
```

SYNOPSIS:

```
(defun move-repeats (list &key (warn t) (test #'eq))
```

16.1.5   permutations/multi-shuffle

[permutations] [Functions]

DESCRIPTION:

```
 Applies the shuffle function a specified number of times to a specified
 list.

 NB: As with the plain shuffle function, the order of the permutations
     returned will always be the same unless the keyword argument
     :fix is set to NIL.
```

ARGUMENTS:

```
 - A sequence.
```

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :start. A zero-based index integer indicating the first element of a
   subsequence to be shuffled. Default = 0.
 - :end. A zero-based index integer indicating the last element of a
   subsequence to be shuffled. Default = the length of the given sequence.
 - :copy. T or NIL to indicate whether the given sequence should be copied
   before it is modified or should be destructively shuffled.
```

```
   T = copy. Default = T.
- :fix. T or NIL to indicate whether the given sequence should always be
  shuffled with the same (fixed) random seed (thus always producing the
  same result). T = fixed seed. Default = T.
- :reset. T or NIL to indicate whether the random state should be reset
  before the function is performed.  T = reset. Default = T.
```

RETURN VALUE:

```
- A sequence.
```

EXAMPLE:

```
;; Simple multi-shuffle with default keywords.
(multi-shuffle '(a b c d e f g) 3)

=> (B A C E D G F)

;; Always returns the same result by default.
(loop repeat 4 do (print (multi-shuffle '(a b c d e f g) 3)))

=>
(B A C E D G F)
(B A C E D G F)
(B A C E D G F)
(B A C E D G F)

;; Set keyword argument :fix to NIL to return different results each time
(loop repeat 4 do (print (multi-shuffle '(a b c d e f g) 3 :fix nil)))

=>
(G C F B D E A)
(A G F B D C E)
(A B D G C F E)
(G C A D E F B)

;; Set keyword arguments :start and :end to shuffle just a subsequence of the
;; given sequence
(loop repeat 4
   do (print (multi-shuffle '(a b c d e f g) 3
                            :fix nil
                            :start 2
                            :end 5)))

=>
(A B D E C F G)
```

```
(A B E C D F G)
(A B E D C F G)
(A B D C E F G)
```

SYNOPSIS:

```
(defun multi-shuffle (seq num-shuffles &key
                       (start 0)
                       (end (length seq))
                       (copy t)
                       (fix t)
                       (reset t))
```

16.1.6   permutations/multi-shuffle-with-perms

[permutations] [Functions]

DESCRIPTION:

Return one permutation from a shuffled list of permutations of the
specified list. The second argument determines how many shuffled
permutations will be in the list from which the resulting permutation is
selected. Similar to the "multi-shuffle" function, but uses the function
"inefficient-permutations" as part of the process.

The <num-shuffles> argument allows the user to always return the same
specific permutation.

NB: This function always uses a fixed random seed and has no optional
    arguments to allow the user to alter that setting.

ARGUMENTS:

- A list.
- An integer that is the number of consecutive shuffles to be collected in
  the list from which the resulting permutation is selected.

RETURN VALUE:

- A list that is a single permutation of the specified list.

EXAMPLE:

```
;; Returns a permutation of a shuffled version of the specified list
(let ((l '(0 1 2 3 4)))
```

```
  (multi-shuffle-with-perms l 7))
```

```
=> (3 1 4 2 0)
```

```
;; Always returns the same result
(loop repeat 4 do (print (multi-shuffle-with-perms '(0 1 2 3 4) 7)))
```

```
=>
(3 1 4 2 0)
(3 1 4 2 0)
(3 1 4 2 0)
(3 1 4 2 0)
```

```
;; Different <num-shuffles> values return different permutations
(loop for i from 0 to 5
   do (print (multi-shuffle-with-perms '(0 1 2 3 4) i)))
```

```
=>
(0 1 2 3 4)
(1 4 2 0 3)
(0 3 1 4 2)
(4 0 2 1 3)
(1 2 3 4 0)
(2 1 3 0 4)
```

SYNOPSIS:

```
(defun multi-shuffle-with-perms (seq num-shuffles)
```

16.1.7   permutations/permutate

[permutations] [Functions]

DESCRIPTION:

 Systematically produce a list of all possible permutations of an original
 list of elements of any type.

 NB: Such lists can quickly become very long, so slippery-chicken
     automatically defaults to outputting the resulting list to a file and
     printing a warning when the results exceed a certain length.

ARGUMENTS:

 - A list with elements of any type.

OPTIONAL ARGUMENTS:

```
- allow-file: whether to allow the results to be written to a file rather
than returned to the interpreter. This will only happen if the length of
the list is > 8. The advantage is that you can then read in the file rather
than slowly regenerate the results if you need them again.
```

RETURN VALUE:

```
A list of lists that are all possible permutations of the original,
specified list.

Interrupts with an error if the method is passed anything but a list.
```

EXAMPLE:

```
;; Simple usage
(permutate '(a b c))

=> ((A B C) (B A C) (A C B) (C A B) (B C A) (C B A))

;; When the list is more than 8 elements long, the resulting permutations are
;; written to a file due to the very high number of results
(permutate '(1 2 3 4 5 6 7 8 9))

=>
WARNING: permutations::permutations: This call will return 362880
results so they are being written to the file
'/tmp/permutations.txt'.
```

SYNOPSIS:

```
(defun permutate (list &optional (allow-file t))
```

16.1.8   permutations/random-rep

[permutations] [Functions]

DESCRIPTION:

```
Return a random non-negative number that is less than the specified
value. An optional argument allows for the random state to be reset.
```

ARGUMENTS:

 – A number.

OPTIONAL ARGUMENTS:

 – T or NIL to indicate whether the random state should be reset before the
   function is performed. T = reset. Default = NIL.

RETURN VALUE:

 A number.

EXAMPLE:

```
;; By default returns a different value each time
(loop repeat 10 do (print (random-rep 5)))

=>
1
3
4
4
3
4
2
0
2
0

;; Setting the optional argument to T resets the random state before
;; performing the function
(loop repeat 10 do (print (random-rep 5 t)))

=>
3
3
3
3
3
3
3
3
3
3
```

SYNOPSIS:

```
(defun random-rep (below &optional (reset nil))
```

16.1.9   permutations/shuffle

[permutations] [Functions]

DESCRIPTION:

Create a random ordering of a given sequence or a subsequence of a given
sequence. By default we used fixed-seed randomness so we can guarantee the
same results each time (perhaps counter-intuitively). So the order of the
permutations returned will always be the same unless keyword argument :fix
is set to NIL.

NB: This function is a modified form of Common Music's shuffle function.

ARGUMENTS:

 - A sequence (list, vector (string)).

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :start. A zero-based index integer indicating the first element of a
  subsequence to be shuffled. Default = 0.
- :end. A zero-based index integer indicating the last element of a
  subsequence to be shuffled. Default = the length of the given sequence.
- :copy. T or NIL to indicate whether the given sequence should be copied
  before it is modified or should be destructively shuffled.
  T = copy. Default = T.
- :fix. T or NIL to indicate whether the given sequence should always be
  shuffled with the same (fixed) random seed (thus always producing the
  same result). T = fixed seed. Default = T.
- :reset. T or NIL to indicate whether the random state should be reset
  before the function is performed.  T = reset. Default = T.
```

RETURN VALUE:

 A list.

EXAMPLE:

```
;; Simple shuffle with default keywords.
(shuffle '(1 2 3 4 5 6 7))

=> (5 4 3 6 7 1 2)
```

```
;; Always returns the same result by default.
(loop repeat 4 do (print (shuffle '(1 2 3 4 5 6 7))))

=>
(5 4 3 6 7 1 2)
(5 4 3 6 7 1 2)
(5 4 3 6 7 1 2)
(5 4 3 6 7 1 2)

;; Set keyword argument :fix to NIL to return different results each time
(loop repeat 4 do (print (shuffle '(1 2 3 4 5 6 7) :fix nil)))

=>
(1 2 6 3 5 4 7)
(1 3 5 2 7 4 6)
(4 7 2 5 1 6 3)
(1 5 3 7 4 2 6)

;; Set the keyword argument :reset to t only at the beginning so we get the
;; same result that time but different (but repeatable) results thereafter.
(loop repeat 3 do
     (print 'start)
     (loop for i below 4
        do (print (shuffle '(1 2 3 4 5 6 7) :reset (zerop i)))))

=>
START
(5 4 3 6 7 1 2)
(4 6 5 2 3 1 7)
(3 4 1 6 5 7 2)
(3 2 7 4 1 6 5)
START
(5 4 3 6 7 1 2)
(4 6 5 2 3 1 7)
(3 4 1 6 5 7 2)
(3 2 7 4 1 6 5)
START
(5 4 3 6 7 1 2)
(4 6 5 2 3 1 7)
(3 4 1 6 5 7 2)
(3 2 7 4 1 6 5)


;; Set keyword arguments :start and :end to shuffle just a subsequence of the
;; given sequence
(loop repeat 4
```

```
  do (print (shuffle '(1 2 3 4 5 6 7)
                     :fix nil
                     :start 2
                     :end 5)))

=>
(1 2 5 4 3 6 7)
(1 2 3 5 4 6 7)
(1 2 4 5 3 6 7)
(1 2 3 4 5 6 7)
```

SYNOPSIS:

```
(defun shuffle (seq &key
                (start 0)
                (end (length seq))
                (copy t)
                (fix t)
                (reset t)
                &aux (width (- end start)))
```

17   sc/samp5

[Modules]

NAME:

```
 samp5

 File:            samp5.lsp

 Class Hierarchy: none, no classes defined

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         clm instrument for sample processing; called by
                  slippery-chicken::clm-play

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   12th June 2004

 $$ Last modified: 11:19:15 Sat Jan  5 2019 CET
```

SVN ID: Id

18   sc/sine

[Modules]

NAME:

samp5

 File: sine.lsp

 Class Hierarchy: none, no classes defined

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: clm instrument for simple sine wave generation. This is
                  used mainly as an example to show how user instruments
                  can be used in clm-play but it may also be useful for a
                  quick sinewave rendition of a piece.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 12th June 2004

 $$ Last modified: 17:19:15 Fri Jan 4 2019 CET

 SVN ID: $Id: sine.lsp 4223 2013-10-29 10:57:09Z medward2 $

19   sc/slippery-chicken-edit

[Modules]

NAME:

slippery-chicken-edit

 File: slippery-chicken-edit.lsp

 Class Hierarchy: named-object -> slippery-chicken

```
Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Post-generation editing methods for the slippery-chicken
                  class.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    April 7th 2012

$$ Last modified:  19:00:17 Thu Sep 24 2020 CEST

SVN ID: $Id$
```

19.1   slippery-chicken-edit/add-arrow-to-events

[slippery-chicken-edit] [Methods]

DATE:

April 9th 2012

DESCRIPTION:

Adds an arrow above the specified notes of a slippery-chicken object,
coupled with text to be printed in the score at the start and end of the
arrow. Can be used, for example, for transitions from one playing state to
another.

If no text is desired, this must be indicated by a space in quotes (" ")
rather than empty quotes ("").

See also the add-arrow method in the event class.

ARGUMENTS:

- A slippery-chicken object.
- A text string for the beginning of the arrow.
- A text string for the end of the arrow.
- A list that is the starting event reference, in the form (bar-number
  event-number). Event numbers count from 1 and include rests and tied
  notes.
- A list that is the end event reference, in the form (bar-number
  event-number).
- The ID of the player to whose part the arrow should be attached.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether or not to print a warning when trying to
  attach an arrow and accompanying marks to a rest.
  T = print warning. Default = NIL.

RETURN VALUE:

 T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :title "mini"
        :ensemble '(((pno (piano :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6)))
                       (2 ((cs4 ds4 fs4 gs4 as4 cs5 ds5 fs5 gs5 as5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q q))
                                :pitch-seq-palette ((1 (2))))))
        :rthm-seq-map '((1 ((pno (1 1 1 1 1 1)))))))))
  (add-arrow-to-events mini "here" "there" '(1 1) '(5 1) 'pno)
  (write-lp-data-for-all mini))
```

SYNOPSIS:

```
(defmethod add-arrow-to-events ((sc slippery-chicken) start-text end-text
                                event1-ref event2-ref player
                                &optional warn-rest)
```

19.2   slippery-chicken-edit/add-auxiliary-notes

[slippery-chicken-edit] [Methods]

DATE:

 May 26th 2016, Edinburgh

DESCRIPTION:

 Analyses phrase by phrase to find the most frequent X notes then changes
 some of them (according to an activity-levels object) to auxiliary notes
 (up a semitone by default). Chords are ignored.

ARGUMENTS:

- the slippery-chicken object

OPTIONAL ARGUMENTS:

keyword arguments
- :players. A single player (symbol) or list of players to process. If NIL
  then all players will be processed.
- :start-bar. An integer indicating the bar in which the processes should
  start. Default = NIL which means we'll start at the beginning.
- :end-bar. An integer indicating the bar in which the processes should
  stop. Default = NIL which means we'll process through to the end
- :verbose. T or NIL to print information about the pitch changes during
  the process. Default = NIL.
- :ignore. A symbol or pitch-object (or list thereof) to represent pitches
  which should not be processed. Default = NIL = process (potentially) all
  pitches.
- :activity-level. Whether to change the note is decided by an
  activity-levels object. This argument determines the 'level' argument used
  in that class's active method. Default = 5 = 50% of the notes will be
  changed. NB The activity-levels object will be reinitialised for each
  phrase so the 50/50 spread might not be obvious.
- :num-notes. The number of notes to process per phrase. We calculate the
  most used pitches in every phrase then use this number of pitches to add
  auxiliaries to. Default = 3.
- :intervals. The intervals that the original pitches will be transposed by
  to create the auxiliary notes. This can be a single value in semitones or
  a list of values. The list does not have to have the same length as
  :num-notes as it will be cycled through when necessary. Default = 1.

RETURN VALUE:

T

SYNOPSIS:

```
(defmethod add-auxiliary-notes ((sc slippery-chicken)
                                &key players start-bar end-bar
                                  verbose ignore (activity-level 5)
                                  (num-notes 3) (intervals 1))
```

19.3   slippery-chicken-edit/add-clef

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Attach a specified clef symbol to a specified clef object within a given
 slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - The ID of the player to whose part the clef symbol is to be added.

 NB: The optional arguments are actually required.

OPTIONAL ARGUMENTS:

 - An integer that is the bar number in which the clef symbol is to be
   placed.
 - An integer that is the event number within the given bar to which the
   clef symbol is to be attached.
 - A symbol that is the clef type to be attached. See the documentation for
   the make-instrument function of the instrument class for a list of
   possible clef types.

RETURN VALUE:

 Returns the new value of the MARKS-BEFORE slot of the given event object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :title "mini"
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6)))
                       (2 ((cs4 ds4 fs4 gs4 as4 cs5 ds5 fs5 gs5 as5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1)))))))))
  (add-clef mini 'vn 3 2 'alto))

  => ((CLEF ALTO))
```

SYNOPSIS:

```
(defmethod add-clef ((sc slippery-chicken) player &optional
                     bar-num event-num clef)
```

19.4   slippery-chicken-edit/add-event-to-bar

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add an event object to a specified bar either at the end of that bar or at
a specified position within that bar.

It's the user's responsibility to make sure the bar is full and
update-slots is called after all post-generation editing such as this.

ARGUMENTS:

- A slippery-chicken object.
- An event object.
- An integer that is the bar number or a list that is the reference to the
  bar in the form '(section sequence bar), where sequence and bar are
  numbers counting from 1)
- The ID of the player to whose part the event should be added.

OPTIONAL ARGUMENTS:

keyword argument:
- :position. NIL or an integer indicating the position in the bar (0-based)
  where the event should be added. If NIL, the new event is placed at the
  end of the bar. Default = NIL.

RETURN VALUE:

T

EXAMPLE:

```
;;; Adding two events to separate bars, once using a bar number with
;;; :position's default to NIL, and once using a bar number reference list with
;;; :position specified as 2. Print the bars after adding to see the changes.

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6)))
                       (2 ((cs4 ds4 fs4 gs4 as4 cs5 ds5 fs5 gs5 as5 cs6))))
```

```
        :set-map '((1 (1 1 1 1 1 1))
                   (2 (2 2 2 2 2 2)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                  :pitch-seq-palette ((1 2 3 4))))
                            (2 ((((2 4) e s s q))
                                  :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))))
                        (2 ((vn (2 2 2 2 2 2)))))))))
  (add-event-to-bar mini (make-event 'cs4 'e) 2 'vn)
  (print-simple (first (get-bar mini 2)))
  (add-event-to-bar mini (make-event 'c4 'q) '(2 2 1) 'vn :position 2)
  (print-simple (first (get-bar mini '(2 2 1)))))
```

```
=>
(2 4): C4 Q, D4 E, F4 S, G4 S, CS4 E
(2 4): CS4 E, DS4 S, C4 Q, FS4 S, GS4 Q
```

SYNOPSIS:

```
(defmethod add-event-to-bar ((sc slippery-chicken) event bar-num-or-ref player
                             &key (position nil)))
```

19.5   slippery-chicken-edit/add-half-beat-rest

[slippery-chicken-edit] [Methods]

DATE:

 August 26th 2016, Edinburgh

DESCRIPTION:

 Calls the method of the same name on each rthm-seq-bar object at the given
 bar number. Adds a half beat rest at the end of the bar. This is a
 destructive method. It will change the time signature, so a 2/4 bar
 becomes a 5/8 bar, with an 1/8th rest at the end. This is done for all the
 players in the ensemble (so that scores can be correctly written)

ARGUMENTS:

 - a slippery-chicken object

OPTIONAL ARGUMENTS:

 - the bar number to modify. Default = 1.

RETURN VALUE:

 T

SYNOPSIS:

(defmethod add-half-beat-rest ((sc slippery-chicken) &optional (bar-num 1)))

19.6   slippery-chicken-edit/add-mark-all-players

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add a specified mark to a specified event in the parts of all players. The
 event can either be specified as a 1-based integer, in which case the mark
 will be attached to the same event in all parts, or as a list of integers,
 in which the mark is attached to different events in the same bar for each
 player, passing from the top of the ensemble downwards.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the bar number or a list of integers that is a
   reference to the bar number in the form (section sequence bar).
 - An integer that is the event to which to attach the specified mark in all
   parts, or a list of integers that are the individual events to which to
   attach the mark in the consecutive players.
 - The mark to be added.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;;; Apply the method twice: Once using an integer to attach the mark to the
;;; same event in all players, and once using a list to attach the mark to
;;; different events in the consecutive players. Print the corresponding marks
;;; slots to see the results.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
```

```
                        (hn (french-horn :midi-channel 2))
                        (vc (cello :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                         (2 ((cl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (add-mark-all-players mini 3 1 'ppp)
  (add-mark-all-players mini '(2 2 1) '(1 2 3) 'fff)
  (loop for i in '(cl hn vc)
     do (print (marks (get-event mini 3 1 i))))
  (loop for i in '(cl hn vc)
     for e in '(1 2 3)
     do (print (marks (get-event mini '(2 2 1) e i)))))

=>
(PPP)
(PPP)
(PPP)
(FFF)
(FFF)
(FFF)
```

SYNOPSIS:

```
(defmethod add-mark-all-players ((sc slippery-chicken)
                                  bar-num event-num mark)
```

19.7   slippery-chicken-edit/add-mark-before-note

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add the specified mark to the MARKS-BEFORE slot of the specified note
 object within the given slippery-chicken object.

 NB: This method counts notes, not events; i.e., rests are not counted.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the bar number in which the mark is to be added.
 - An integer that is the NOTE number to which the mark is to be added (not
   the event number; i.e., rests are not counted).
 - The ID of the player to which the mark is to be added.
 - The mark to be added.

RETURN VALUE:

 Returns the new value of the MARKS-BEFORE slot of the given event object.

EXAMPLE:

```
;;; The method adds the mark to the specified note, not event. Add the mark to
;;; note 2, print the MARKS-BEFORE slots of events 2 (which is a rest) and 3.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1)))))))
  (add-mark-before-note mini 3 2 'vn 'ppp)
  (print (marks-before (get-event mini 3 2 'vn)))
  (print (marks-before (get-event mini 3 3 'vn))))

=>
NIL
(PPP)
```

SYNOPSIS:

```
(defmethod add-mark-before-note ((sc slippery-chicken)
                                       bar-num note-num player mark)
```

19.8   slippery-chicken-edit/add-mark-to-event

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add the specified mark to the MARKS slot of the specified event within the
 given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the bar number to which the mark is to be added.
 - An integer that is the event number in the specified bar to which the
   mark is to be added.
 - The ID of the player to which to add the mark.
 - The mark to add.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; Add a mark to an event object then read the value of the MARKS slot of that
;;; event to see the result
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))))))))
  (add-mark-to-event mini 3 2 'vn 'ppp)
  (marks (get-event mini 3 2 'vn)))

=> (PPP)
```

SYNOPSIS:

```
(defmethod add-mark-to-event ((sc slippery-chicken) bar-num event-num player
                              mark)
```

19.9   slippery-chicken-edit/add-mark-to-note

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add the specified mark to the specified note of a given slippery-chicken
object.

NB: This method counts notes, not events; i.e., not rests.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the bar number to which to add the mark
- An integer that is the note number two which to add the mark. This is
  1-based, and counts notes not events; i.e., not rests. If a list, then it
  is of the form (note-num chord-note), where chord-note is 1-based and
  counts from the lowest note up.
- The ID of the player to whose part the mark is to be added.
- The mark to add.

RETURN VALUE:

Returns T.

EXAMPLE:

```
;;; Add a mark to a note in a bar with a rest. Print the corresponding event
;;; object to see the result.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))))))))
  (add-mark-to-note mini 3 2 'vn 'ppp)
  (print (marks (get-event mini 3 2 'vn)))
  (print (marks (get-event mini 3 3 'vn))))

=>
NIL
(PPP)
```

SYNOPSIS:

```
(defmethod add-mark-to-note ((sc slippery-chicken)
                             bar-num note-num player mark)
```

19.10   slippery-chicken-edit/add-marks-sh

[slippery-chicken-edit] [Methods]

DATE:

 27-Jun-2011

DESCRIPTION:

 Add marks in a somewhat more free list form, with the option of
 implementing a user-defined shorthand.

ARGUMENTS:

 - A slippery-chicken object.
 - A list of lists containing the players, bar and note refs, and marks to
   be added. The first element of each contained list will be the ID of the
   player to whose part the marks are to be added followed by a pattern of
   <mark bar-number note-number> triplets, or if a mark is to be added
   repeatedly then <mark bar note bar note... >. A mark can be a string or a
   symbol.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - For marks given as symbols, the user can supply a shorthand table that
   will expand an abbreviation, such as sp, to the full mark name, such as
   short-pause. This table takes the form of a simple Lisp association list,
   e.g.: '((al aeolian-light)
           (ad aeolian-dark)
           (wt "WT")
           (h harm))
 - :warn. T or NIL to indicate whether to print a warning for unrecognized
   marks. T = print warning. Default = T.
 - :verbose. T or NIL to indicate whether the method is to print verbose
   feedback about each mark added to the Listener. T = print feedback.
   Default = NIL.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))
                            (va (1 1 1 1 1 1))))))))
  (add-marks-sh mini
                '((vn a 1 1 1 2 3 1 s 2 1 2 2 2 5)
                  (va pizz 1 3 2 3 sp 3 1))
                :shorthand '((sp short-pause))
                :verbose t))
```

```
=> NIL
```

SYNOPSIS:

```
(defmethod add-marks-sh ((sc slippery-chicken) player-data
                         &key shorthand (warn t) verbose)
```

19.11   slippery-chicken-edit/add-marks-to-note

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add one or more specified marks to a specified note within a given
slippery-chicken object.

NB: This method counts notes, not events; i.e., rests are not counted.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the bar number to which the mark or marks should to be
  added.
- An integer that is the note within the specified bar to which the mark or
  marks should be added.
- The ID of the player to whose part the mark or marks should be added.
- The mark or marks to add.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; Add several marks to one note, then print the corresponding MARKS slot to
;;; see the difference.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e (e) e e (e) e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))
                            (va (1 1 1 1 1 1)))))))))
  (add-marks-to-note mini 2 3 'va 'a 's 'lhp 'pizz)
  (print (marks (get-note mini 2 3 'va))))

=> (PIZZ LHP S A)
```

SYNOPSIS:

```
(defmethod add-marks-to-note ((sc slippery-chicken) bar-num note-num
                              player &rest marks)
```

19.12   slippery-chicken-edit/add-marks-to-notes

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add the specified mark or marks to a consecutive sequence of multiple notes
 within the given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer or a list consisting of two numbers to indicate the start
   bar/note. If this is an integer, all notes in this bar will receive the
   specified mark or marks. If this is a two-number list, the first number

  determines the bar, the second the note within that bar.
- An integer or a list consisting of two numbers to indicate the end
  bar/note. If this is an integer, all notes in this bar will receive the
  specified mark or marks. If this is a two-number list, the first number
  determines the bar, the second the note within that bar.
- The ID of the player or players to whose parts the mark or marks should
  be attached. This can be a single symbol or a list.
- T or NIL to indicate whether the mark should be added to the MARKS slot
  or the MARKS-BEFORE slot of the given events objects.
- The mark or marks to be added.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; This example calls the method twice: Once using the single-integer
;;; indication for full bars, with one instrument and one mark; and once using
;;; the bar/note reference lists for more specific placement, a list of several
;;; players that should all receive the marks, and multiple marks to add.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))
                            (va (1 1 1 1 1 1)))))))))
  (add-marks-to-notes mini 2 3 'vn nil 'lhp)
  (add-marks-to-notes mini '(1 3) '(2 2) '(vn va) nil 's 'a))

=> T
```

SYNOPSIS:

```
(defmethod add-marks-to-notes ((sc slippery-chicken) start end players before
                                  &rest marks)
```


19.13   slippery-chicken-edit/add-pitches-to-chord

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add specified pitches to an existing chord object.

ARGUMENTS:

 - The slippery-chicken object which contains the given chord object.
 - The ID of the player whose part is to be affected.
 - An integer that is the number of the bar that contains the chord object
   that is to be modified.
 - An integer that is the number of the note that is the chord object to be
   modified.
 - The pitches to be added. These can be pitch objects or any data that can
   be passed to make-pitch, or indeed lists of these, as they will be
   flattened.

RETURN VALUE:

 The chord object that has been changed.

EXAMPLE:

```
(let* ((ip-clone (clone +slippery-chicken-standard-instrument-palette+)))
  (set-slot 'chord-function 'chord-fun1 'guitar ip-clone)
  (let* ((mini
          (make-slippery-chicken
           '+mini+
           :instrument-palette ip-clone
           :ensemble '(((gtr (guitar :midi-channel 1))))
           :set-palette '((1 ((e3 f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5 d5 e5 f5
                               g5 a5 b5 c6 d6 e6))))
           :set-map '((1 (1)))
           :rthm-seq-palette
           '((1 ((((4 4) e e e e e e e e))
                 :pitch-seq-palette ((1 (2) 3 (4) 5 (6) 7 (8))))))
           :rthm-seq-map '((1 ((gtr (1)))))))))
    (print (get-pitch-symbols
            (pitch-or-chord (get-event mini 1 2 'gtr))))
    (add-pitches-to-chord mini 'gtr 1 2 'cs4 'ds4)
    (print (get-pitch-symbols
            (pitch-or-chord (get-event mini 1 2 'gtr))))))

=>
(E3 G3 B3)
(E3 G3 B3 CS4 DS4)
```

SYNOPSIS:

```
(defmethod add-pitches-to-chord ((sc slippery-chicken) player bar-num note-num
                                 &rest pitches)
```

19.14   slippery-chicken-edit/add-player

[slippery-chicken-edit] [Methods]

DATE:

 October 30th 2018, Heidhausen

DESCRIPTION:

 Add a player to the ensemble and piece slots/objects of the given
 slippery-chicken object. This clones the section/sub-section structure of
 the slippery-chicken object, creating all the necessary data structures for
 empty bars in the right meter, tempo etc.

 NB As this method may be called several times successively, it's the
 caller's duty to call (update slots sc) in order to have timing and other
 data updated correctly.

ARGUMENTS:

 - the slippery-chicken object
 - a player object or symbol ID for the new player

OPTIONAL ARGUMENTS:

 - a symbol ID for an existing instrument in the instrument-palette (the next
   argument). This is actually required unless the default of 'computer is
   acceptable or a player object is passed as second argument.
 - an instrument-palette object in which the instrument exists. Default is
   the standard palette.
 - the midi-channel for the new player
 - the microtones-midi-channel for the new player

RETURN VALUE:

 the new player object from the ensemble slot of the slippery-chicken object

SYNOPSIS:

```
(defmethod add-player ((sc slippery-chicken) player
                       &optional (instrument 'computer)
                         (instrument-palette
                          +slippery-chicken-standard-instrument-palette+)
                         ;; MDE Tue Jul 14 19:08:15 2020, Heidhausen
                         (midi-channel 1)
                         (microtones-midi-channel -1))
```

19.15   slippery-chicken-edit/add-tuplet-bracket-to-bar

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add a tuplet bracket (with number) to a specified bar in a slippery-chicken
object. This method adds only one tuplet bracket of one tuplet type
(triplet, quintuplet etc.) at a time.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar to which the tuplet bracket is
  to be added.
- The ID of the player to whose part the tuplet bracket is to be added.
- The bracket info defining the tuplet bracket to be added. This takes the
  form of a three-element list specifying tuplet value, number of the event
  (zero-based) on which the bracket is to begin, and number of the event on
  which the bracket is to end, e.g. '(3 0 2).

OPTIONAL ARGUMENTS:

T or NIL to indicate whether all existing tuplet brackets in the given bar
are to be deleted first. T = delete. Default = NIL>

RETURN VALUE:

T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+sc-object+
        :ensemble '(((va (viola :midi-channel 2))))
        :set-palette '((1 ((c3 d3 e3 f3 g3 a3 b3 c4 d4))))
```

```
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((3 4) (te) - te te - { 3 te ts+ts te }
                                  - fs fs fs fs fs -))
                              :pitch-seq-palette ((1 2 3 4 5 6 7 8 9 8)))))
        :rthm-seq-map '((1 ((va (1 1 1)))))))))
  (add-tuplet-bracket-to-bar mini 1 'va '(3 0 2))
  (add-tuplet-bracket-to-bar mini 2 'va '(5 7 11))
  (add-tuplet-bracket-to-bar mini 3 'va '(3 3 4) t)
  (add-tuplet-bracket-to-bar mini 3 'va '(3 5 6)))

=> T
```

SYNOPSIS:

```
(defmethod add-tuplet-bracket-to-bar ((sc slippery-chicken) bar-num player
                                      bracket-info
                                      &optional (delete-all-tuplets-first nil))
```

19.16   slippery-chicken-edit/add-tuplet-brackets-to-beats

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Add the specified tuplet brackets (and numbers) to the specified event
 objects in the specified bars within the given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - The ID of the player to whose part the tuplet brackets are to be added.
 - A list of 4-element sublists that is the bracket info. Each sublist must
   consist of: the number of the bar to which the bracket is to be added;
   the number that is the tuplet type (3 = triplet, 5 = quintuplet etc.);
   the zero-based number of the event where the bracket is to begin; the
   zero-based number that is the number of the event where the bracket is to
   end; e.g. '((2 3 0 5) (3 3 0 3) (5 5 0 4))

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether all existing tuplet bracket info in the given
 bars is to first be deleted. T = delete. Default = NIL.

RETURN VALUE:

```
 NIL.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+sc-object+
        :ensemble '(((va (viola :midi-channel 2))))
        :set-palette '((1 ((c3 e3 g3 c4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((3 4) - te te te - - fs fs fs fs fs -
                                  - 28 28 28 28 28 28 28 -))
                                :pitch-seq-palette ((1 2 3 4 1 2 3 4 1 2 3 4 1
                                                     2 3)))))
        :rthm-seq-map '((1 ((va (1 1 1)))))))))
  (add-tuplet-brackets-to-beats mini 'va '((1 3 0 2) (2 5 3 7) (3 7 8 14))))

=> NIL
```

SYNOPSIS:

```
(defmethod add-tuplet-brackets-to-beats
    ((sc slippery-chicken) player bracket-info
     &optional (delete-all-tuplets-first nil))
```

19.17   slippery-chicken-edit/auto-accidentals

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Automatically determine which notes in each bar need accidentals and which
don't.

This method also places cautionary accidentals (in parentheses) based on
how many notes back the last occurrence of that note/accidental combination
appeared in the bar. The first optional argument to the method allows the
user to specify how many notes back to look.

NB: As both cmn-display and write-lp-data-for-all call respell-notes by
    default, that option must be set to NIL for this method to be
    effective (see below).

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

- An integer that is the number of notes back to look when placing
  cautionary accidentals in parentheses. If the last occurrence of a given
  repeated note/accidental combination was farther back than this number,
  the accidental will be placed in the score in parentheses.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((fs4 gs4 as4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 2 1 2 3 2)))))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (auto-accidentals mini 4)
  (cmn-display mini :respell-notes nil))

=> NIL
```

SYNOPSIS:

```
(defmethod auto-accidentals ((sc slippery-chicken) &optional
                             (cautionary-distance 3)
                             ignore1 ignore2)
```

19.18   slippery-chicken-edit/auto-beam

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Automatically places indications for beam start- and end-points (1 and 0)
 in the BEAMS slot of the corresponding event objects.

 By default, this method determines the start and end indications for beams

on the basis of the beat found in the time signature, but the user can
specify a different beat basis using the first optional argument.

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

- NIL, an integer that is a power-of-two rhythmic duration, or the
  alphabetic representation of such a rhythm to specify the beat basis for
  setting beams (e.g. 4 or 'h).
- T or NIL to indicate whether the method is to check whether an exact beat
  of rhythms can be found for each beat of the bar. If T, a warning will be
  printed when an exact beat cannot be found for each beat of the bar.
  Default = T.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
;; Auto-beam the events of the given slippery-chicken object on the basis of a ;
;; half note:                                      ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((fs4 gs4 as4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 2 1 2 3 2)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (auto-beam mini 'h))

  => NIL
```

SYNOPSIS:

```
(defmethod auto-beam ((sc slippery-chicken) &optional (beat nil) (check-dur t))
```

19.19   slippery-chicken-edit/auto-clefs

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Automatically create clef changes in the specified player's or players'
part or parts by adding the appropriate clef symbol to the MARKS-BEFORE
slot of the corresponding event object.

This routine will only place clef symbols that are present in the given
instrument object's CLEFS slot.

This method is called automatically by cmn-display and
write-lp-data-for-all, with the delete-clefs option set to T.

NB: While this routine generally does a good job of putting the proper
    clefs in place, it will get confused if the pitches in a given player's
    part jump from very high to very low (e.g. over the complete range of
    the piano).

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :verbose. T or NIL to indicate whether the method is to print feedback
  about its operations to the Listener. T = print feedback. Default = NIL.
- :in-c. T or NIL to indicate whether the pitches processed are to be
  handled as sounding or written pitches. T = sounding. Default = T.
- :players. A list containing the IDs of the players whose parts are to be
  to have clefs automatically added.
- :delete-clefs. T or NIL to indicate whether the method should first
  delete all clef symbols from the MARKS-BEFORE slots of all event objects
  it is processing before setting the automatic clef changes.
- :delete-marks-before. T or NIL to indicate whether the MARKS-BEFORE slot
  of all event objects processed should first be set to NIL.
  T = set to NIL. Default = NIL.

RETURN VALUE:

Returns T

EXAMPLE:

```
;;; Straightforward usage applied to just the VC player ;
(let ((mini
```

```
      (make-slippery-chicken
       '+mini+
       :ensemble '(((vn (violin :midi-channel 1))
                    (vc (cello :midi-channel 2))))
       :tempo-map '((1 (q 60)))
       :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
       :set-map '((1 (1 1 1 1)))
       :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                               :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
       :rthm-seq-map '((1 ((vn (1 1 1 1))
                           (vc (1 1 1 1)))))))))
  (auto-clefs mini :players '(vc)))

  => T
```

SYNOPSIS:

```
(defmethod auto-clefs ((sc slippery-chicken)
                       &key verbose in-c players
                       (delete-clefs t)
                       (delete-marks-before nil))
```

19.20   slippery-chicken-edit/auto-slur

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Automatically add slurs to note groups in the specified measure range of a
given player's part.

This method places slurs above all consecutive notes between rests. If a
value is specified for :end-bar and the last event in the end bar is not a
rest, the final sequence of attacked notes in that bar will not be
slurred.

NB: Slurs will automatically stop at repeated pitches. Staccato marks will
not stop the auto-slurring process but staccatos can be removed (see
below).

ARGUMENTS:

- A slippery-chicken object.
- A player ID or list of player IDs for the parts in which the slurs are to
  be placed.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :start-bar. An integer that is the first bar in which to automatically
  place slurs.
- :end-bar. An integer that is the last bar in which to automatically place
  slurs.
- :rm-slurs-first. T or NIL to indicate whether to first remove existing
  slurs from the specified region. NB: If you already have slur marks
  attached to events, setting this to NIL can produce unwanted results
  caused by orphaned beg-slur or end-slur marks.  T = remove existing slurs
  first. Default = T.
- :rm-staccatos. T or NIL to indicate whether to first remove existing
  staccato, tenuto, and accented staccato marks from the specified
  region. T = remove staccatos. Default = NIL.
- :over-accents. T or NIL. Default = T.
- :verbose. T or NIL to indicate whether to print feedback from the process
  to the Lisp listener. T = print. Default = NIL.
```

RETURN VALUE:

```
A list of sublists, each of which contains the start and end event, plus
the number of notes under the slur, for each slur added.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c3 d3 e3 f3 g4 a3 b3
                               c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) - e e - (s) e.
                                     - s s e - - s (s) s s -))
                                 :pitch-seq-palette ((1 2 3 4 5 6 7 8 9))
                                 :marks (a 4))))
        :rthm-seq-map '((1 ((vn (1 1 1)))))))))
  (auto-slur mini 'vn
             :start-bar 1
             :end-bar 2))
```

SYNOPSIS:

```
(defmethod auto-slur ((sc slippery-chicken) players
```

```
                            &key start-bar end-bar
                            (rm-slurs-first t)
                            (rm-staccatos t)
                            ;; 5.4.11
                            (over-accents t)
                            verbose)
```

19.21   slippery-chicken-edit/bars-to-sc

[slippery-chicken-edit] [Functions]

DESCRIPTION:

Take a list of rthm-seq-bars and add them to a new or existing
slippery-chicken object. It works in three different ways:
   1. Create a new slippery chicken object with one player and one section.
   2. Add new parts to an already existing section in an already exisiting
      slippery chicken object.
   3. Add a new section to an already exisiting part in and already
      exisiting object.

In the second case, we assume you are creating a slippery chicken object
part by part with this function. It is not currently possible to
add a part like this in the middle of the score--the new part will be added
to the end of the last group of the ensemble (bottom of score) so make sure
to add parts in the order you want them.

In the third case, you can add a new section to an exisitng part, but you
cannot add a new section to a part that does not exist.

NB Bear in mind that if you want to use midi-play, then the events in the
   bars will need to have their midi-channel set (e.g. via make-event).
   It's also the caller's responsibility to ensure that the parts added have
   the same time-signature structure as any existing part.

ARGUMENTS:

 - A list of rthm-seq-bars

OPTIONAL ARGUMENTS:

keyword arguments:
 - :sc. Either an existing slippery-chicken object or nil if one should be
   created automatically. If nil, the following three arguments must be

    specified, otherwise they will be ignored. Default = NIL.
  - :sc-name. The name (symbol) for the slippery-chicken object to be
    created. This will become a global variable. Default = '*auto*.
  - :player. The name (symbol) of the player to create. Default =
    'player-one. (Remember that Lilypond has problems with player names
     with numbers in them :/)
  - :instrument. The id (symbol) of an already existing instrument in the
     instrument-palette. Default = 'flute.
  - :update. Whether to call update-slots on the new slippery-chicken
    object (to update timing info. etc.). Default = T.
  - :section-id. The section id. Default = 1.
  - :tempo. The tempo in BPM for the whole slippery-chicken object (piece).
  - :midi-channels. A 2-element list specifying the midi-channels for the
    chromatic and microtonal notes of the player. This will not change
    channels of the events in the bars but is assumed to correspond to those.

RETURN VALUE:

 A slippery-chicken object.

SYNOPSIS:

```
(defun bars-to-sc (bars &key sc (sc-name '*auto*) (player 'player-one)
                        (midi-channels '(1 2))
                        (instrument-palette
                         +slippery-chicken-standard-instrument-palette+)
                        (tempo 60) (instrument 'flute) (section-id 1)
                        (update t))
```

19.22   slippery-chicken-edit/change-pitch

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Change the pitch of a specified event to a new specified pitch. The new
 pitch is not required to be a member of the current set.

 NB The new pitch is the sounding pitch if a transposing instrument, unless
 optional argument is set to T.

 NB In the case of ties, this doesn't update the following 'tied-to' notes
 but see the documentation for the change-pitches method below (or call
 check-ties, which fixes them too).

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the bar number in which the pitch is to be changed.
- An integer that is the number of the note in the specified bar whose
  pitch is to be changed (1-based).
- The ID of the player for whom the pitch is to be changed.
- A note-name symbol that is the new pitch.

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether the written or sounding pitch should be
 changed. Default = NIL (sounding).

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (change-pitch mini 1 3 'vc 'fs3))

=> T
```

SYNOPSIS:

```
(defmethod change-pitch ((sc slippery-chicken) bar-num note-num player
                         new-pitch &optional written)
```

19.23   slippery-chicken-edit/change-pitches

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Change the pitches of the specified event objects for a given player to the
 specified new pitches.

If the new pitches are passed as a simple flat list, the method will just
change the pitch of each consecutive attacked event object (with NIL
indicating no change), moving from bar to bar as necessary, until all of
the specified new pitches are used up. Also, if a flat list is passed, each
new pitch specified will be applied to each consecutive attacked note;
i.e., ties don't count as new pitches.

Tied-to events are left with the old pitch information, which is of course
a potential problem. When generating scores though, we usually call
respell-notes, which calls check-ties, which corrects spellings of tied-to
notes and therefore in effect changes those notes too. So generally, we
don't have to worry about this, but if you explicitly tell slippery chicken
not to respell notes, you'll need to call check-ties with the first
optional argument as T.

Also see the documentation in the bar-holder class for the method of the
same name.

NB As various methods (e.g. transposition, change-pitch(es)) in various
classes (slippery-chicken, bar-holder, rthm-seq-bar, rthm-seq) may change
pitch information and midi-channels stored in events, chords, pitches,
etc. it is recommended to call the slippery-chicken reset-midi-channels
method before calling midi-play, if any of those methods have been
used. This way midi channels for pitches belonging to particular players
will use channels originally allocated in the ensemble for chromatic and
microtonal pitches, no matter where they originated or how they were
changed.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player whose part is to be modified.
- An integer that is the number of the first bar whose pitches are to be
  modified.
- A list note-name symbols and NILs, or a list of lists of note-name
  symbols and NILs, which are the new pitches. If a simple flat list, see
  the comment in the description above. If a list of lists, each sub-list
  will represent a full bar; e.g., (change-pitches bh 'vla 5 '((g3 gs4) nil
  (nil nil aqf5))) will change the pitches in bars 5 and 7 (for the player
  'vla), whereas bar six, indicated by nil, wouldn't be changed; similarly
  the first two notes of bar 7, being nil, will also not be changed, but
  note 3 will.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :use-last-octave. T or NIL to indicate whether or not each consecutive
  new pitch listed will automatically take the most recent octave number
  specified; e.g. '((a3 b g cs4)). T = use last octave number. Default = T.
- :marks. A list of marks to be added to the events objects. This option
  can only be used in conjunction with the simple flat list of pitches. In
  this case the list of pitches and list of marks must be the same length
  and correspond to each other item by item. Sub-lists can be used to add
  several marks to a single event. NB: See cmn.lsp::get-cmn-marks for the
  list of recognised marks. If NIL, no marks will be added. Default = NIL.
- :written.  T or NIL to indicate whether these are the written or sounding
  notes for a transposing instrument. Default = NIL.
- :warn.  If there are more pitches in the given list than there are events
  in the slippery-chicken structure, issue a warning, unless NIL.
  Default = T.
```

RETURN VALUE:

```
 If the new pitches are passed as a simple flat list, the method returns
 the number of the bar in which the pitches were changed;
 otherwise returns T.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1 1 1)))))))))
  (change-pitches mini 'vc 2 '((fs3 gs3 as3)))
  (change-pitches mini 'vc 3 '((nil nil fs3 gs as ds fs gs)
                               nil
                               (cs4 ds fs))))

=> T
```

SYNOPSIS:

```
(defmethod change-pitches ((sc slippery-chicken) player start-bar new-pitches
                           &key (use-last-octave t) marks written
                                (warn t))
```

19.24   slippery-chicken-edit/change-time-sig

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Force a change of the time-sig associated with a specified bar.

 NB: This method does not check to see if the rhythms in the bar add up to a
     complete bar in the new time-sig.

 Also see rthm-seq-bar (setf time-sig).

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar whose time signature should be
   changed or a list that is a reference to the bar whose time signature is
   to be changed in the format (section sequence bar).
 - The new signature in the format (number-of-beats beat-unit).

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; Changing two time signatures; once using the integer bar reference, the
;;; second time using the list reference to the bar number.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                 :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1))))))))
  (change-time-sig mini 2 '(3 8))
  (change-time-sig mini '(1 1 1) '(5 8)))

=> T
```

SYNOPSIS:

```
(defmethod change-time-sig ((sc slippery-chicken) bar-num-or-ref new-time-sig)
```

19.25   slippery-chicken-edit/consolidate-all-notes

[slippery-chicken-edit] [Methods]

DESCRIPTION:

A convenience method which just calls the consolidate-notes method from the
rthm-seq-bar class for all the bars specified in the arguments.

ARGUMENTS:

- the slippery-chicken object
- the first bar number in which consolidation should take place
- the last bar number in which consolidation should take place
  (inclusive). If NIL then the process will run until the last bar.
- A list of the IDs of the players to whose parts the consolidation should
  be applied. Can also be a single symbol.

RETURN VALUE:

A list of the rthm-seq-bar objects that were modified. See map-over-bars
for more details.

SYNOPSIS:

```
(defmethod consolidate-all-notes ((sc slippery-chicken)
                                  &optional (start-bar 1) end-bar players
                                  ;; MDE Fri Sep 21 09:08:09 2018 -- added
                                  ;; consolidate-notes' optional args
                                    check-dur beat (auto-beam t))
```

19.26   slippery-chicken-edit/consolidate-all-rests

[slippery-chicken-edit] [Methods]

DESCRIPTION:

A convenience method which just calls the consolidate-rests method from the
rthm-seq-bar class for all the bars specified in the arguments.

ARGUMENTS:

- the slippery-chicken object
- the first bar number in which consolidation should take place
- the last bar number in which consolidation should take place

```
    (inclusive).  If NIL then the process will run until the last bar.
- A list of the IDs of the players to whose parts the consolidation should
  be applied. Can also be a single symbol.
```

OPTIONAL ARGUMENTS:

```
T or NIL to indicate whether the method should print a warning to the Lisp
listener if it is mathematically unable to consolidate the rests. T = print
warning. Default = NIL.
```

RETURN VALUE:

```
- A list of the rthm-seq-bar objects that were modified.  See map-over-bars
 for more details.
```

SYNOPSIS:

```
(defmethod consolidate-all-rests ((sc slippery-chicken)
                                  &optional (start-bar 1) end-bar players warn)
```

19.27   slippery-chicken-edit/delete-bars

[slippery-chicken-edit] [Methods]

DESCRIPTION:

```
Delete a sequence of consecutive bars from the given slippery-chicken
object.
```

```
NB This might delete rehearsal letters, instrument changes (and maybe other
things) attached to a bar/event.
```

ARGUMENTS:

```
- A slippery-chicken object.
- An integer that is the first bar to delete.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :num-bars. An integer that is the number of consecutive bars, including
  the start-bar, to delete. This argument cannot be used simultaneously
  with :end-bar
- :end-bar. An integer that is the number of the last of the consecutive
```

  bars to delete. This argument cannot be used simultaneously with
  :num-bars.
- :print. Print feedback of the process to the Listener, including a
  print-simple of the bars deleted.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (delete-bars mini 2 :end-bar 3)
  (delete-bars mini 2 :num-bars 1))

  => T
```

SYNOPSIS:

```
(defmethod delete-bars ((sc slippery-chicken) start-bar
                        &key num-bars end-bar print)
```

19.28   slippery-chicken-edit/delete-clefs

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Delete the clef symbol held in the MARKS-BEFORE slot of the specified event
 object within the given slippery-chicken object.

ARGUMENTS:

 NB: The optional arguments are actually required.
 - A slippery-chicken object.
 - The ID of the player from whose part the clef symbol is to be deleted.

- An integer that is the number of the bar from which the clef symbol is to
  be deleted.
- An integer that is the number of the event object within the specified
  from whose MARKS-BEFORE slot the clef symbol is to be deleted. This is a
  1-based index but counts rests and ties.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (auto-clefs mini)
  (delete-clefs mini 'vc 1 3))

  => NIL
```

SYNOPSIS:

```
(defmethod delete-clefs ((sc slippery-chicken) &optional
                          player bar-num event-num)
```

19.29   slippery-chicken-edit/delete-events

[slippery-chicken-edit] [Methods]
DATE:

 21-Jul-2011 (Pula)

DESCRIPTION:

 Turn notes into rests by setting the IS-REST slots of the specified
 consecutive event objects within the given slippery-chicken object to T.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the first bar for which the notes are to
   be changed to rests.
 - An integer that is the index of the first event object within the
   specified start bar for which the IS-REST slot is to be changed to
   T. This number is 1-based and counts rests and ties.
 - An integer that is the number of the last bar for which the notes are to
   be changed to rests.
 - An integer that is the index of the last event object within the
   specified end bar for which the IS-REST slot is to be changed to T. This
   number is 1-based and counts rests and ties. If NIL, apply the change to
   all events in the given bar.

OPTIONAL ARGUMENTS:

 - A list of the IDs of the players whose parts are to be modified. If NIL,
   apply the method to the parts of all players.
 - T or NIL to indicate whether to consolidate resulting consecutive rests
   into one longer rest each. T = consolidate. Default = T.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (delete-events mini 2 2 3 nil 'vc))

  => T
```

SYNOPSIS:

```
(defmethod delete-events ((sc slippery-chicken) start-bar start-event end-bar
                          end-event &optional players (consolidate-rests t))
```

19.30   slippery-chicken-edit/delete-rehearsal-letter

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Delete the rehearsal letter from a specified bar of on or more specified
players' parts by setting the REHEARSAL-LETTER slot of the corresponding
rthm-seq-bar object to NIL.

NB: This deletes the given rehearsal letter without resetting and
    re-ordering the remaining rehearsal letters.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar from which the rehearsal letter
  is to be deleted. NB: The rehearsal letter for a given bar is internally
  actually attached to the previous bar. The number given here is the
  number from the user's perspective, but the change will be reflected in
  the bar with the number specified -1.

OPTIONAL ARGUMENTS:

- A list consisting of the IDs of the players from whose parts the
  rehearsal letter is to be deleted.

RETURN VALUE:

Returns the deleted rehearsal letter from the last player which had one
attached.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1 1 1)))))
        :rehearsal-letters '(2 4 6))))
```

```
(delete-rehearsal-letter mini 2 '(vc)))

=> NIL
```

SYNOPSIS:

```
(defmethod delete-rehearsal-letter ((sc slippery-chicken) bar-num
                                    &optional players)
```

19.31   slippery-chicken-edit/delete-slur

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Delete a slur mark that starts on a specified note within a specified bar
of a specified player's part by deleting the BEG-SL and END-SL marks from
the corresponding event objects.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar from which the slur is to be
  deleted.
- An integer that is the number of the note on which the slur to be deleted
  starts within the given bar. This number counts tied-notes but not
  rests.
- The ID of the player from whose part the slur is to be deleted.

RETURN VALUE:

Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8))
                                :marks (slur 1 8))))
```

```
        :rthm-seq-map '((1 ((vc (1 1 1 1 1 1)))))))))
 (delete-slur mini 1 1 'vc)
 (delete-slur mini 3 1 'vc))

 => NIL
```

SYNOPSIS:

```
(defmethod delete-slur ((sc slippery-chicken) bar-num note-num player)
```

19.32   slippery-chicken-edit/double-events

[slippery-chicken-edit] [Methods]

DATE:

```
 20-Jul-2011 (Pula)
```

DESCRIPTION:

Copy the specified events from one player to the corresponding bars of one
or more other players.

NB: Although partial bars can be copied from the source player, the entire
    bars of the target players are always overwritten, resulting in rests
    in those segments of the target players' bars that do not contain the
    copied material. This method thus best lends itself to copying into
    target players parts that have rests in the corresponding bars.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player from whose part the events are to be copied.
- The ID or a list of IDs of the player or players into whose parts the
  copied events are to be placed.
- An integer that is the number of the first bar from which the events are
  to be copied.
- An integer that is the number of the first event to be copied from the
  specified start bar. This number is 1-based and counts rests and ties.
- An integer that is the number of the last bar from which the events are
  to be copied.
- NIL or an integer that is the number of the last event to be copied from
  the specified end bar. This number is 1-based and counts rests and
  ties. If NIL, all events from the given bar will be copied.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :transposition. A positive or negative number that is the number of
  semitones by which the copied material is to be first transposed. This
  number can be a decimal number, in which case the resulting pitches will
  be rounded to the nearest microtone (if the current tuning environment is
  capable of microtones).
- :consolidate-rests. T or NIL to indicate whether resulting consecutive
  rests should be consolidated each into one longer rest.
  T = consolidate. Default = T.
- :update. T or NIL to indicate whether to update the slots of the given
  slippery-chicken object after copying. T = update. Default = T.
- :auto-beam. T or NIL to indicate whether to call auto-beam on the doubled
  notes. Default = T.
- :pitches. T or NIL to indicate whether pitch information should be copied
  over. This would usually be T but some methods might just want to
  duplicate events as rests (e.g. orchestrate). Default = T.
```

RETURN VALUE:

```
Returns T
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((bsn (bassoon :midi-channel 1))
                     (tbn (tenor-trombone :midi-channel 2))
                     (vlc (cello :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) (w)))))
                            (2 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((bsn (1 1 1 1 1 1))
                            (tbn (1 1 1 1 1 1))
                            (vlc (2 2 2 2 2 2)))))))
  (double-events mini 'vlc '(bsn tbn) 2 3 4 2)
  (double-events mini 'vlc 'bsn 5 1 5 nil :transposition 3.5))

  => T
```

SYNOPSIS:

```
(defmethod double-events ((sc slippery-chicken) master-player doubling-players
                          start-bar start-event end-bar end-event
                          &key transposition (consolidate-rests t) (update t)
                             (pitches t) (auto-beam t))
```

19.33   slippery-chicken-edit/double-player-inverted

[slippery-chicken-edit] [Methods]

DATE:

 July 14th 2020, Heidhausen

DESCRIPTION:

 Add a new player to the ensemble/piece, double the events of an existing
 player, then use the invert method for each bars within the given range to
 turn rests into notes and vice-versa. To choose new notes we use curves to
 limit high and low, using the sets (from the map) used to generate the
 existing voices.

ARGUMENTS:

 - the slippery-chicken object
 - the ID of a new player (symbol)
 - the instrument for the new player (must be in the instrument-palette)
 - the ID of the existing player whose events are inverted (symbol)
 - a curve for the maximum upper notes
 - a curve for the maximum lower notes. Both these curves can have any
   arbitrary x range but the y values should use MIDI note numbers or note
   symbols. Note that the x range will be stretched over the number of bars
   in the piece, not the given bar range, so that the bar number can be used
   for interpolation.
 - a curve to decide whether to actually play a note in the new part or
   not. This can also have any arbitrary x range and will also be stretched
   over the number of bars in the piece, but the y range should be from 0 to
   10 as the interpolated value for the notes in each bar is passed to an
   activity-levels object to determine whether to play or not.

OPTIONAL ARGUMENTS:

 keyword arguments:
 :start-bar :start-event :end-bar :end-event. These are passed to
 double-events. See that method for details but note that the defaults for
 the end-* arguments are NIL which in turn default to the last bar/event.

:reset-midi-channels. T or NIL to reset all events in the slippery-chicken
object to the original midi channels of the parent player. Default = T.

RETURN VALUE:

the (modified) slippery-chicken object

SYNOPSIS:

```
(defmethod double-player-inverted ((sc slippery-chicken) new-player new-ins
                                    existing-player upper-curve lower-curve
                                    &key (start-bar 1) (start-event 1)
                                      (activity-curve '(0 10 100 10))
                                      end-bar end-event
                                      (reset-midi-channels t) (update-slots t)
                                      (midi-channel 1)
                                      (microtones-midi-channel -1))
```

19.34   slippery-chicken-edit/enharmonic-spellings

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Change the pitch of specified event objects to their enharmonic
equivalents.

This takes as its second argument a list of lists, each of which consists
of the ID of the player whose part is to be altered and a series of
bar-number/event-number pairs, where (2 3) indicates that the pitch of the
third event of the second bar is to be changed to its enharmonic
equivalent.

Pitches within chords are specified by following the bar number with a
2-item list consisting of the event number and the number of the pitch
within the chord, counting from low to high, where (2 (2 4)) indicates that
the fourth pitch from the bottom of the chord located in the second event
object of bar 2 should be changed to its enharmonic equivalent.

An optional T can be included to indicate that the written pitch is to be
changed, but not the sounding pitch, as in (cl (3 4 t)).

NB: In order for this method to work, the :respell-notes option of
    cmn-display and write-lp-data-for-all must be set to NIL.

ARGUMENTS:

```
- A slippery-chicken object.
- The list of changes to be made, in the format '((player changes...)),
  e.g.:

  '((cl (3 3 t) (3 4 t))
    (pn (2 (2 4)))
    (vc (1 1) (1 3) (1 4) (1 6)))
```

RETURN VALUE:

 Returns T.

EXAMPLE: SYNOPSIS:

(defmethod enharmonic-spellings ((sc slippery-chicken) corrections)

19.35   slippery-chicken-edit/enharmonics

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Change the sharp/flat pitches of a specified region of a specified player's
 part to their enharmonic equivalent.

 NB: This method only affects pitches that already have sharp/flat
     accidentals. It does not affect "white-key" notes (e.g. C-natural =
     B-sharp etc.)

 NB: As the cmn-display and write-lp-data-for-all methods call
     :respell-notes by default, this option must be explicitly set to NIL for
     this method to be effective.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer or a 2-item list of integers that indicates the first bar in
   which the enharmonics are to be changed. If an integer, the method will
   be applied to all sharp/flat pitches in the bar of that number. If a
   2-item list of integers, these represent '(bar-number note-number). The
   note number is 1-based and counts ties. If nil, it defaults to 1.
 - An integer or a 2-item list of integers that indicates the last bar in
   which the enharmonics are to be changed. If an integer, the method will
   be applied to all sharp/flat pitches in the bar of that number. If a
   2-item list of integers, these represent '(bar-number note-number). The

   note number is 1-based and counts ties. If nil it defaults to (num-bars sc)
- The ID of the player or players whose part is to be changed. If nil, the
  method will be applied to all players.

OPTIONAL ARGUMENTS:

 keyword arguments
- :written. T or NIL to indicate whether to change written-only pitches or
  sounding-only pitches. T = written-only. Default = T.
- :pitches. NIL or a list of note-name symbols. If NIL, all sharp/flat
  pitches in the specified region will be changed to their enharmonic
  equivalents. If a list of one or more note-name symbols, only those
  pitches will be affected.
- :force-naturals. T or NIL to indicate whether to force "natural" note
  names that contain no F or S in their name to convert to their enharmonic
  equivalent (ie, B3 = CF4). NB double-flats/sharps are not implemented so
  this will only work on F/E B/C.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (pn (piano :midi-channel 2))
                     (vn (violin :midi-channel 3))))
        :set-palette '((1 ((cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) - e e e e - - e e e e -))
                                :pitch-seq-palette ((1 (2) 3 4 (5) 6 (7) 8)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (pn (1 1 1 1 1))
                            (vn (1 1 1 1 1)))))))))
  (enharmonics mini 1 2 'vn)
  (enharmonics mini 2 3 'pn :pitches '(cs4 ds4))
  (enharmonics mini 3 4 'cl :written nil)
  (enharmonics mini nil nil nil))

=> T
```

SYNOPSIS:

```
(defmethod enharmonics ((sc slippery-chicken) start end players
                        &key (written t) pitches force-naturals)
```

19.36   slippery-chicken-edit/fast-microtone-to-chromatic

[slippery-chicken-edit] [Methods]

DATE:

 4 September 2019, London

DESCRIPTION:

 Round microtonal pitches to their nearest chromatic pitch if the time
 between successive events is less than or equal to a specified threshold in
 seconds. This method is intended to make playing fast pasages in microtonal
 music easier for human players.

ARGUMENTS:

 - The slippery chiocken object containing the pitches to be tested.
 - The player or players to be tested

OPTIONAL ARGUMENTS:

 - start-bar: the first bar to start testing in. Default = 1
 - end-bar: the last bar to test in. Default = (num-bars sc)
 - threshold: the time in seconds between successive events that determines
 whether or not a pitch should be rounded to its nearest chromatic. Default =
 (fast-leap-threshold sc).

RETURN VALUE:

 A list containing the number of rounded pitches per instrument.

EXAMPLE:

```
(let ((mini (make-slippery-chicken
                '+mini+
                :title "Your Title Here"
                :composer "Your Name Here"
                :ensemble '(((flt (flute :midi-channel 1 :microtones-midi-channel 2))))
                :staff-groupings '(1)
                :tempo-map '((1 (q 60)))
```

```
                    :set-palette '((set1 ((fqs2 b2 dqs4 aqf4 dqs5 e5 a5 d6))))
                    :set-map '((1 (set1 set1 set1 set1 set1 set1)))
                    :rthm-seq-palette
                    '((seq1 ((((4 4) q - s s s s - - e e - q))
                            :pitch-seq-palette (1 2 3 4 5 6 7 8))))
                    :rthm-seq-map
                    '((1 ((flt (seq1 seq1 seq1 seq1 seq1 seq1)))))))))
        (fast-microtone-to-chromatic mini nil :threshold 10)
        (cmn-display mini))
```

SYNOPSIS:

```
(defmethod fast-microtone-to-chromatic ((sc slippery-chicken) players
                                    &key
                                      (start-bar 1)
                                      end-bar
                                      ;; threshold in secs
                                      threshold)
```

19.37   slippery-chicken-edit/force-artificial-harmonics

[slippery-chicken-edit] [Methods]

DESCRIPTION:

For string scoring purposes only: Transpose the pitch of the given event
object down two octaves and add the harmonic symbol at the perfect fourth.

If this results in a fingered pitch (or even a touched perfect fourth) that
is out of the range of the instrument, a warning will be printed to the
Listener, the pitch will not be transposed, and the harmonic diamond will
not be added.

NB This method assumes the player doesn't change instrument during the bars
to be processed.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player whose part is to be changed.
- An integer that is the number of the first bar in which artificial
  harmonics are to be created.
- An integer that is the number of the first event in that bar that is to
  be changed into an artificial harmonic.
- An integer that is the number of the last bar in which artificial
  harmonics are to be created.

OPTIONAL ARGUMENTS:

```
 - An integer that is the number of the first event in that bar that is to
   be changed into an artificial harmonic. If no end-event is specified, all
   event objects in the last bar will be changed to artificial harmonics.
 - T or NIL to trigger warnings when forcing would take a fingered pitch out
   of the instrument's range, or when we encounter chords or rests
 - T or NIL to force natural-harmonics to become artificial. Default = T.
```

RETURN VALUE:

```
 Returns T.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 f4 b4 e5 a5 d6 g7 c8))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vn (1 1 1)))))))
  (force-artificial-harmonics mini 'vn 2 3 3 2))

=> T
```

SYNOPSIS:

```
(defmethod force-artificial-harmonics ((sc slippery-chicken) player start-bar
                                       start-event end-bar
                                       &optional end-event (warn t)
                                         (naturals t))
```

19.38   slippery-chicken-edit/force-harmonics

[slippery-chicken-edit] [Methods]

DATE: DESCRIPTION: ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod force-harmonics ((sc slippery-chicken) player start-bar
                            &key (start-event 1) end-bar end-event warn
                              (tolerance 15))
```

19.39 slippery-chicken-edit/force-in-range

[slippery-chicken-edit] *[Methods]*

DATE:

 October 19th 2018, Heidhausen

DESCRIPTION:

 Force all pitches (and chords) to be within the range of the instruments who
 play them. Of course this is usually a part of the slippery-chicken pitch
 selection algorithm but sometimes we change pitches (algorithmically in
 particular) and this might bring them out of range. Notes that are out of
 range will be transposed up and down the required number of octaves. This
 applies to single pitches in chords also. Instrument changes will be
 respected on a bar-by-bar basis. See also the three force-in-range methods
 in instrument.lsp

ARGUMENTS:

 - the slippery-chicken object
 - a symbol or list of symbols for the players to be processed. Default = NIL
   = all players.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start-bar. The bar to start at. Default = NIL = 1
 - :end-bar. The bar to end at. Default = NIL = last bar in piece/object

RETURN VALUE:

 the modified slippery-chicken object

SYNOPSIS:

(defmethod force-in-range ((sc slippery-chicken) players &key start-bar end-bar)

19.40 slippery-chicken-edit/force-natural-harmonics

[slippery-chicken-edit] *[Methods]*

**DATE: DESCRIPTION: ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE:
EXAMPLE: SYNOPSIS:**

```
(defmethod force-natural-harmonics ((sc slippery-chicken) player start-bar
                                    &optional start-event end-bar end-event
                                      (warn t) (tolerance 15))
```

19.41   slippery-chicken-edit/force-rest-bars

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Delete all notes from the specified bars and replace them with full-bar
 rests.

 NB: The start-bar and end-bar index numbers are inclusive

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the first bar to change to a full bar of
   rest.
 - An integer that is the number of the last bar to change to a full bar of
   rest. If NIL then we process all bars.
 - A list containing the IDs of the players in whose parts the full-bar
   rests are to be forced. If NIL then all players will be processed.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate that only bars containing exclusively rests should be
   processed.

RETURN VALUE:

 Returns the number of bars processed (integer).

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 e4 g4 b4 d5 f5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1)))
```

```
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e))
                              :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))
                            (va (1 1 1 1 1 1))
                            (vc (1 1 1 1 1 1))))))))))
  (force-rest-bars mini 3 5 '(vn vc)))

=> NIL
```

SYNOPSIS:

```
(defmethod force-rest-bars ((sc slippery-chicken) start-bar end-bar players
                            &optional only-if-all-rests)
```

19.42   slippery-chicken-edit/make-hammer-friendly

[slippery-chicken-edit] [Methods]

DATE:

October 4th 2019

DESCRIPTION:

Yamaha Disklavier pianos can't play a note immediately after a note-off. The
hammer needs a little time to return to its 'off' position before it can be
used to restrike a note. Tests show this time to be around 60
milliseconds. of course this will be instrument-dependent, so a little
experimentation might be needed.

This method will go through each event and decrement those events which
repeat a note on the same MIDI channel which start < 'min-time' after the
end-time of the previous event.

ARGUMENTS:

- the slippery-chicken object
- either a single symbol or a list of symbols for the players to be
  processed

OPTIONAL ARGUMENTS:

keyword arguments
- :start-bar. The bar at which to begin processing (integer). Default = 1.
- :end-bar. The bar at which to end processing (integer). Default = NIL =

```
    last bar
 - :min-gap. The minimum time the hammer needs to return, in
   milliseconds. Default = 60.
 - :min-dur. The minimum duration of a note in milliseconds, after reducing
   duration. Default = 50 milliseconds.
```

RETURN VALUE:

 A list of the number of notes affected per player.

SYNOPSIS:

```
(defmethod make-hammer-friendly ((sc slippery-chicken) players
                                 &key
                                    (warn t)
                                    (start-bar 1)
                                    end-bar
                                    (min-gap 60)
                                    (min-dur 50))
```

19.43   slippery-chicken-edit/map-over-bars

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Apply the specified method/function to the bars (all rthm-seq-bar objects)
 of one or more players' parts in the given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - A number that is the first bar to which the function should be
   applied. Default = NIL in which case 1 will be used.
 - A number that is the last bar to which the function should be
   applied. Default = NIL in which case all bars will be processed.
 - A list of the IDs of the players to whose parts the function should be
   applied. Can also be a single symbol. If NIL then all players will be
   processed.
 - The method or function itself. This can be a user-defined function or the
   name of an existing method or function. It should take at least one
   argument, a rthm-seq-bar, and any other arguments as supplied.

OPTIONAL ARGUMENTS:

- Any additional argument values the specified method/function may
  take or require. See the thin method below for an example that uses
  additional arguments.

RETURN VALUE:

- A list of the rthm-seq-bar objects that were modified. NB This might be
  a long list, and, depending on your Lisp implementation, formatting of the
  bars might cause Lisp to appear to 'hang'.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                 (3 ((sax ((3 alto-sax) (4 tenor-sax)))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h e (s) (s) e+s+s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))))
                        (3 ((sax (1 1 1 1 1))))))))
  (print (map-over-bars mini 1 nil nil #'consolidate-notes nil 'q)))

=>
(
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: T, bar-num: 1,
[...]
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: T, bar-num: 2,
[...]
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: T, bar-num: 3,
[...]
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: T, bar-num: 4,
[...]
)
```

SYNOPSIS:

```
(defmethod map-over-bars ((sc slippery-chicken) start-bar end-bar players
                          function &rest further-args)
```

19.44 slippery-chicken-edit/map-over-events

[slippery-chicken-edit] [Methods]

DATE:

 1 September 2019, London

DESCRIPTION:

 A simple method that mimics the behaviour of map-over-bars but on events
 instead. At its heart is a next-event loop. Use this method to apply an
 arbitrary function to every event within a specific range of bars.

ARGUMENTS:

 - A slippery chicken object
 - A number that is the first bar to which the function should be
   applied. Default = NIL in which case 1 will be used.
 - A number that is the last bar to which the function should be
   applied. Default = NIL in which case all bars will be processed.
 - A list of the IDs of the players to whose parts the function should be
   applied. Can also be a single symbol. If NIL then all players will be
   processed.
 - The method or function itself. This can be a user-defined function or the
   name of an existing method or function. It should take at least one
   argument, an event, and any other arguments as supplied.

OPTIONAL ARGUMENTS:

 - Any additional argument values the specified method/function may
   take or require. See the thin method below for an example that uses
   additional arguments.

RETURN VALUE:

 - A list containing the number of events changed per instrument

EXAMPLE:

```
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
         :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
```

```
                                       (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                       (3 ((sax ((3 alto-sax) (4 tenor-sax))))))
              :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
              :set-map '((1 (1 1 1 1 1))
                         (2 (1 1 1 1 1))
                         (3 (1 1 1 1 1)))
              :rthm-seq-palette '((1 ((((4 4) h e (s) (s) e+s+s))
                                      :pitch-seq-palette ((1 2 3)))))
              :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                              (2 ((sax (1 1 1 1 1))))
                              (3 ((sax (1 1 1 1 1)))))))))
        (print (is-rest (get-event mini (num-bars mini) 2 'sax)))
        (print (map-over-events mini 1 nil nil #'force-rest))
        (is-rest (get-event mini (num-bars mini) 3 'sax)))

=> NIL
   (105)
   T
```

SYNOPSIS:

```
(defmethod map-over-events ((sc slippery-chicken) start-bar end-bar players
                            function &rest further-args)
```

19.45   slippery-chicken-edit/map-over-events-aux

[slippery-chicken-edit] [Methods]

DATE:

 3 September 2019, London

DESCRIPTION:

 Auxilliary function for map-over-notes and map-over-events. Saves
 duplicating code.

ARGUMENTS:

 - A slippery chicken object
 - A number that is the first bar to which the function should be
   applied. Default = NIL in which case 1 will be used.
 - A number that is the last bar to which the function should be
   applied. Default = NIL in which case all bars will be processed.
 - A list of the IDs of the players to whose parts the function should be

   applied. Can also be a single symbol. If NIL then all players will be
   processed.
 - T or NIL to indicate whether or not rests will be ignored.
   T = attacked notes only (i.e. no rests), NIL = include rests.
 - The method or function itself. This can be a user-defined function or the
   name of an existing method or function. It should take at least one
   argument, an event, and any other arguments as supplied.
 - Any additional argument values the specified method/function may
   take or require. See the thin method below for an example that uses
   additional arguments.

RETURN VALUE:

 - A list containing the number of events changed per instrument

SYNOPSIS:

```
(defun map-over-events-aux (sc start-bar end-bar players attacked-notes-only
                              function further-args)
```

19.46   slippery-chicken-edit/map-over-notes

[slippery-chicken-edit] [Methods]

DATE:

 1 September 2019, London

DESCRIPTION:

 A simple method that mimics the behaviour of map-over-bars but on notes
 instead. At its heart is a next-event loop. Use this method to apply an
 arbitrary function to every note within a specific range of bars.
 NB This works on NOTES only, rests will be skipped!

ARGUMENTS:

 - A slippery chicken object
 - A number that is the first bar to which the function should be
   applied. Default = NIL in which case 1 will be used.
 - A number that is the last bar to which the function should be
   applied. Default = NIL in which case all bars will be processed.
 - A list of the IDs of the players to whose parts the function should be
   applied. Can also be a single symbol. If NIL then all players will be
   processed.

- The method or function itself. This can be a user-defined function or the
  name of an existing method or function. It should take at least one
  argument, a note event (can be a single pitch or a chord), and any other
 arguments as supplied.

OPTIONAL ARGUMENTS:

- Any additional argument values the specified method/function may
  take or require. See the thin method below for an example that uses
  additional arguments.

RETURN VALUE:

- A list containing the number of notes changed per instrument

EXAMPLE:

```
(let* ((mini
         (make-slippery-chicken
          '+mini+
          :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
          :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                    (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                    (3 ((sax ((3 alto-sax) (4 tenor-sax))))))
          :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
          :set-map '((1 (1 1 1 1 1))
                     (2 (1 1 1 1 1))
                     (3 (1 1 1 1 1)))
          :rthm-seq-palette '((1 ((((4 4) h e (s) (s) e+s+s))
                                  :pitch-seq-palette ((1 2 3)))))
          :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                          (2 ((sax (1 1 1 1 1))))
                          (3 ((sax (1 1 1 1 1))))))))
       (print (is-chord (get-note mini (num-bars mini) 3 'sax)))
       (print (map-over-notes mini 1 nil nil #'add-pitches 'c4 'd4))
       (is-chord (get-note mini (num-bars mini) 3 'sax)))

=> NIL
   (45)
   3
```

SYNOPSIS:

```
(defmethod map-over-notes ((sc slippery-chicken) start-bar end-bar players
                           function &rest further-args)
```

19.47   slippery-chicken-edit/map-over-sequenzes

[slippery-chicken-edit] [Methods]

DATE:

 February 2nd 2018, Heidhausen

DESCRIPTION:

 Apply a function to each sequenz object in a slippery-chicken piece. Note
 that as a sequenz is a subclass of rthm-seq (and bar-holder) and that
 rthm-seq has its rthm-seq-bars in the bars slot, we've got access to all
 kinds of data which we can switch on in the function passed here,
 e.g. player-section-ref, rsp-id

ARGUMENTS:

 - the slippery-chicken object
 - the player(s) to process (single or list of symbols). If nil then all
   players will be processed.
 - the function to apply to each sequenz. This should accept a sequenz object
   as its first argument.

OPTIONAL ARGUMENTS:

 &rest: further arguments to be passed to the given function

RETURN VALUE:

 T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
```

```
                                  (2 ((((4 4) h h))
                                      :pitch-seq-palette ((1 2)))))
         :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                             (vc (1 1 1 1 1))))
                          (2 ((hn (2 2 2 2 2))
                             (vc (2 2 2 2 2))))
                          (3 ((hn (1 1 1 1 1))
                             (vc (1 1 1 1 1)))))))))
  (map-over-sequenzes mini 'hn
                      #'(lambda (seq) (print (player-section-ref seq)))))
=>
(1 HN)
(1 HN)
(1 HN)
(1 HN)
(1 HN)
(2 HN)
(2 HN)
(2 HN)
(2 HN)
(2 HN)
(3 HN)
(3 HN)
(3 HN)
(3 HN)
(3 HN)
T
```

SYNOPSIS:

```
(defmethod map-over-sequenzes ((sc slippery-chicken) players function
                               &rest further-args)
```

19.48   slippery-chicken-edit/move-clef

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Move a specified clef from a specified event object to another.

NB: As the :auto-clefs option of cmn-display and write-lp-data-for all
    first deletes all clefs before automatically placing them, this
    argument must be set to NIL. The auto-clefs method can be called
    outside of the cmn-display or write-lp-data-for-all methods instead.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the given clef is
  located.
- An integer that is the number of the event object in the given bar to
  which the given clef is attached.
- An integer that is the number of the bar to which the given clef is
  to be moved (this can be the same bar).
- An integer that is the number of the event object in the new bar to
  which the given clef is to attached.
- The ID of the player in whose part the clef is to be moved.

RETURN VALUE:

Returns the value of the MARKS-BEFORE slot of the event object to which the
clef is moved.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (auto-clefs mini)
  (move-clef mini 1 6 1 8 'vc)
  (cmn-display mini :auto-clefs nil))
```

SYNOPSIS:

```
(defmethod move-clef ((sc slippery-chicken) from-bar from-event
                      to-bar to-event player)
```

19.49 slippery-chicken-edit/move-events

[slippery-chicken-edit] [Methods]

DATE:

20-Jul-2011 (Pula)

DESCRIPTION:

Move a specified sequence of consecutive event objects from one player to another, deleting the events from the source player.

NB: Although partial bars can be moved from the source player, the entire
    bars of the target players are always overwritten, resulting in rests
    in those segments of the target players' bars that do not contain the
    moved material. This method thus best lends itself to moving into
    target players parts that have rests in the corresponding bars.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the source player.
- The ID of the target player.
- A number that is the first bar from which events are to be moved.
- A number that is the first event within the start-bar that is to be
  moved.
- A number that is the last bar from which events are to be moved.
- A number that is the last event within the end-bar that is to be
  moved.

OPTIONAL ARGUMENTS:

keyword arguments:
- :transposition. A positive or negative number that is the number of
  semitones by which the copied material is to be first transposed. This
  number can be a decimal number, in which case the resulting pitches will
  be rounded to the nearest microtone (if the current tuning environment is
  capable of microtones).
- :consolidate-rests. T or NIL to indicate whether resulting consecutive
  rests should be consolidated each into one longer rest.
  T = consolidate. Default = T.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
```

```
      '+mini+
      :ensemble '(((bn (bassoon :midi-channel 1))
                   (vc (cello :midi-channel 2))))
      :tempo-map '((1 (q 60)))
      :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
      :set-map '((1 (1 1 1 1)))
      :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                 :pitch-seq-palette ((1 2 3 4 5 6 7 8))))
                          (2 ((((4 4) (w)))))))
      :rthm-seq-map '((1 ((bn (1 1 1 1))
                          (vc (2 2 2 2))))))))))
 (move-events mini 'bn 'vc 2 3 3 2)
 (move-events mini 'bn 'vc 4 1 4 2 :transposition 4.5))

 => T
```

SYNOPSIS:

```
(defmethod move-events ((sc slippery-chicken) from-player to-player
                        start-bar start-event end-bar end-event
                        &key transposition (consolidate-rests t))
```

19.50   slippery-chicken-edit/move-rehearsal-letter

[slippery-chicken-edit] [Methods]

DATE:

 March 15th 2017, Edinburgh

DESCRIPTION:

 Move a rehearsal letter for those players who have one at <from-bar> to
 <to-bar>

ARGUMENTS:

 - a slippery-chicken object
 - the bar number (integer) to move the letter from
 - the bar number (integer) to move the letter to

RETURN VALUE:

 The last rehearsal-letter that was seen (in the lowest player in the
 ensemble)

SYNOPSIS:

```
(defmethod move-rehearsal-letter ((sc slippery-chicken) from-bar to-bar)
```

19.51 slippery-chicken-edit/note-add-bracket-offset

[slippery-chicken-edit] [Methods]

DESCRIPTION:

For CMN only: Adjust the position, lengths, and angles of the tuplet
bracket attached to a specified event object.

NB: The bracket data is stored in the BRACKET slot of the first event
    object of a given tuplet figure.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the tuplet bracket is
  located.
- An integer that is the event to which the tuplet bracket is
  attached. Tuplet brackets are attached to the first event object of a
  given tuplet figure.
- The ID of the player in whose part the tuplet bracket is located.

OPTIONAL ARGUMENTS:

keyword arguments:
NB: At least one of these arguments must be set in order to create a
    change.
- :dx. A positive or negative decimal number to indicate the horizontal
  offset of the entire bracket.
- :dy. A positive or negative decimal number to indicate the vertical
  offset of the entire bracket.
- :dx0. A positive or negative decimal number to indicate the horizontal
  offset of the left corner of the bracket.
- :dy0.A positive or negative decimal number to indicate the vertical
  offset of the left corner of the bracket.
- :dx1. A positive or negative decimal number to indicate the horizontal
  offset of the right corner of the bracket.
- :dy1. A positive or negative decimal number to indicate the vertical
  offset of the right corner of the bracket.
- :index. An integer that indicates which bracket of a nested bracket on
  the same event is to be affected. 0 = outermost bracket, 1 = first nested
  bracket, etc. Default = 0.

RETURN VALUE:

 Returns a list of the bracket start/end indicator and the tuplet value
 followed by the offset values passed to the keyword arguments.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((f3 g3 a3 b3))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((2 4) { 3 te te te } q ))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vc (1))))))))
  (note-add-bracket-offset mini 1 1 'vc
                           :dx -.1 :dy -.3
                           :dx0 -.1 :dy0 -.4
                           :dx1 .3 :dy1 -.1))

=> (1 3 -0.1 -0.3 -0.1 -0.4 0.3 -0.1)
```

SYNOPSIS:

```
(defmethod note-add-bracket-offset ((sc slippery-chicken)
                                    bar-num note-num player
                                    &key (dx nil) (dy nil)
                                    (dx0 nil) (dy0 nil)
                                    (dx1 nil) (dy1 nil)
                                    (index 0))
```

19.52   slippery-chicken-edit/octavise-repeated-notes

[slippery-chicken-edit] [Methods]

DATE:

 July 13th 2020, Heidhausen (though taken from earlier projects)

DESCRIPTION:

 Transpose fast repeated notes by given intervals, typically, as in piano
 music, by an octave. Note that if there are several fast repeated notes then

there will be multiple transpositions e.g. (c4 c4 c4 c4 c4) would become by default (c4 c5 c4 c5 c4).

ARGUMENTS:

- the slippery-chicken object
- the player (symbol) or players (list) to process

OPTIONAL ARGUMENTS:

keyword arguments:
- :start-bar. The bar to start processing at (integer). Default = 1.
- :end-bar. The bar to stop processing at (inclusive). Default = NIL = last bar of the piece.
- :intervals. A list of semitone intervals. Default = '(12) = only transpose up one octave. A list of any length may be used for variation; it will be used cyclically.
- :threshold. The maximum time difference in seconds between events that will result in the 'octavisation'. Default = the slippery-chicken object's fast-leap-threshold (0.125 seconds by default).
- :verbose. Print bar numbers where processing takes place. Default = NIL.
- :whole-chords. T or NIL to transpose whole chords or just the repeated notes in chords (could result in unplayable chords if NIL). Default = T = transpose whole chord.
- :check-ties. Whether to call the check-ties method, which as a side-effect makes sure tied notes are of the same pitch. Default = T.

RETURN VALUE:

the (modified) slippery-chicken object.

SYNOPSIS:

```
(defmethod octavise-repeated-notes ((sc slippery-chicken) players
                                    &key (start-bar 1) end-bar verbose
                                      (intervals '(12)) (whole-chords t)
                                      (check-ties t)
                                      (threshold (fast-leap-threshold sc)))
```

19.53   slippery-chicken-edit/orchestrate

[slippery-chicken-edit] [Methods]

DATE:

November 2nd 2018, Heidhausen

DESCRIPTION:

'Orchestrate' the chords and single pitches in an existing slippery-chicken
object. This involves mapping the existing players' pitch events
onto a group of new players (a 'combo'). The procedure is fairly
complex and by no means perfect (in fact it may exit with an error,
especially if your requirements are :strict (see below)), but it works
like this:

First, we assume that your existing music is more horizontal/harmonic than
linear/melodic, particularly because the mapping process treats each 'chord'
separately and potentially maps each chord onto different players---the
choice of whether to change ensemble is decided by an activity-levels object
by default (see :combo-change-fun below) but also by the number of notes in
any given chord and whether it's possible to play any subsequent chord on
the same instruments. The method also assumes that each 'player' in the
existing slippery-chicken object has a sequence of one or more chords which
can be separated into 'phrases' using the get-phrases method; that the new
ensemble uses player IDs that don't exist in the original ensemble (though
any player can use the same instruments, of course); and that using the
player IDs from the new ensemble we provide lists of sublists of possible
'combos' that should be used to play the existing chords. These combos
should range from 2 players to however many we'll need to play the existing
chords. We pass these combos to the method and/or allow them to be created
by the 'lotsa-combos' ensemble method (see :combos below)

Whether a chord is playable by a given combo is determined by the range of
the chord and the instruments of any given combo, along with those
instruments' abilities to play chords and/or microtones (if the chord is
microtonal). When trying to determine whether a given combo can play a
chord, we permutate the possible instrument orderings to see if the pitches
are in range. If :artificial-harmonics is T we also try for such on notes
normally considered too high for any given string instrument. If we can't
find a combo for any given chord we'll keep track of the best match and if
our :relax argument allows, use that instead of backing out completely.

One of the complexities we face, after determining that a chord is playable
by a particular combo, is whether the players are free (i.e. not already
playing) at the time the chord needs to be played. We take ties into account
of course, but one limitiation, presently, is that in order be deemed 'free'
each instrument of the combo must have a complete bar free for the start and
end of any given chord.

As we select the current instrument using the instrument-change-map slot of

the slippery-chicken object (which may have been changed after the original
data was generated) this method works with players who double on two or more
instruments.

ARGUMENTS:

- the slippery-chicken object whose existing players and event data will be
  used to orchestrate onto new players
- a new ensemble, the players/instruments of which will be used for the new
  orchestration. (This shouldn't be a nested ensemble but that's a rarity.)
  This can either be an ensemble object or a list structure that can be made
  into one (as used for most calls to make-slippery-chicken).
- a list of player IDs indicating the original music to be re-orchestrated.

OPTIONAL ARGUMENTS:

keyword arguments:
- :combos. A list of lists indicating the player groupings to be used to
  orchestrate the events. Default = NIL whereupon the <new-ensemble> will be
  used to generate groupings via the 'lotsa-combos' ensemble class method.
- :add-more-combos. T or NIL to indicate whether the :combos slot should be
  expanded via the 'lotsa-combos' method. If T then whatever is passed to
  :combos will nevertheless be retained after the expansion. Default = NIL.
- :start-bar. The bar number to start orchestrating. Default = NIL which
  will in turn default to 1
- :end-bar. The bar number to stop orchestrating (inclusive). Default = NIL
  which will in turn default to the last bar of the slippery-chicken object.
- :auto-beam. T or NIL to indicate whether to call auto-beam in the new
  parts. Default = T.
- :chords. Whether individual instruments capable of playing chords should
  attempt to do so. This may determine whether a given combo is determined
  to be capable of playing one of the original slippery-chicken object's
  chords, and also whether under some circumstances some notes will be
  doubled across different instruments. Default = T.
- :combo-change-fun. A function which should return T or NIL to indicate
  whether we should change combo on any given event. This should take two
  arguments (the combo as a list of player IDs and an assoc-list of all
  possible combos (as created by the organise-combos method using the
  <combos> argument given here) plus two further optional arguments which
  are free to define. The function should also provide "reset" functionality
  or do nothing if one of the two required arguments is NIL. Default =
  combo-change? (defined in event.lsp).
- :verbose. T or NIL to turn on the printing of orchestration decisions as
  the algorithm is run.
- :artificial-harmonics. T or NIL to allow the selection of string
  artificial harmonics in order to reach high notes outwith normal ranges.

- :sticky-stats. T or NIL to indicate whether the statistics collected and
  returned by the method should be reset or retained between calls. Default
  = NIL = reset.
- :relax. An integer between 0 and 3 inclusive to indicate how strict the
  algorithm should when allocating chord pitches to instruments. The values
  have the following meanings with degrees of relaxedness ascending:
  0: each pitch in the chord has to be playable by at least one instrument
  in the combo; no instrument can sit this chord out but instruments can
  play chords that include notes found in other instruments.
  1. instruments may sit out a chord if a chord-playing instrument can fill
  in the notes (all notes of chord must still be played)
  2. notes can be left out and instruments can sit out a chord
  3. whole chords can be left out

RETURN VALUE:

Five statistics values representing the number of chords where: 1. all
pitches & every player used; 2. all pitches but some players tacit; 3. some
pitches and instruments missing; 4. chords completely left out; 5. total
chords attempted

EXAMPLE:

```
(let* ((mini (make-slippery-chicken
               '+mini+
               :ensemble '(((flt (flute :midi-channel 1))))
               :staff-groupings '(1)
               :tempo-map '((1 (q 60)))
               :set-palette '((set1 ((fs2 b2 d4 a4 d5 e5 a5 d6)))
                              (set2 ((b2 fs3 d4 e4 a4 d5 e5 a5 d6))))
               :set-map '((1 (set1 set1 set2 set1 set1 set2)))
               :rthm-seq-palette
               '((seq1 ((((4 4) q q q q))
                        :pitch-seq-palette (1 2 1 3)))
                 (seq2 ((((4 4) (e) e q h))
                        :pitch-seq-palette (1 2 3))))
               :rthm-seq-map '((1 ((flt (seq1 seq1 seq2 seq1 seq1 seq2)))))))
       (new-ens (make-ensemble 'new-ens
                                '((vln (violin :midi-channel 2))
                                  (clr (b-flat-clarinet :midi-channel 3))))))
  (orchestrate mini new-ens 'flt)
  (cmn-display mini))
```

=> T

SYNOPSIS:

```
(let (successes)                        ; for stats
  (defmethod orchestrate ((sc slippery-chicken) new-ensemble original-players
                          &key
                             combos add-more-combos start-bar end-bar verbose
                             (artificial-harmonics t) (relax 0) (auto-clefs t)
                             (auto-beam t) (combo-change-fun #'combo-change?)
                             (chords t) sticky-stats)
```

19.54 slippery-chicken-edit/pause-last

[slippery-chicken-edit] [Methods]

DATE:

 23 September 2016, London

DESCRIPTION:

 Add a pause mark to the last note of every (any) player's part. It defaults
 to the last bar but this can be changed with the :bar-num keyword arg. You
 also have the option to change the bar line type in the same bar using the
 :bar-line keyword arg. See (change-bar-line-type) for details.

ARGUMENTS:

 - the slippery chicken object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :players. The player (symbol) or list of players that will be
   effected. Default = NIL = all players.
 - :bar-num. The bar number to add the pause mark. Default = last bar of
   piece.
 - :bar-line. The bar line type of the bar we'll add a pause to (symbol). Can
   be one of 'normal 'double-bar 'final-double 'begin-repeat
   'begin-end-repeat or 'end-repeat. Default = NIL = no change.
 - :pause. The type of pause to add (symbol). Can be one of 'pause
   'long-pause or 'short-pause. Default = 'pause

RETURN VALUE:

 - the number of pause marks added

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((pno (piano :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q q))
                                 :pitch-seq-palette ((1 (2)))))))
        :rthm-seq-map '((1 ((pno (1 1 1 1 1 1 1 1)))))))))
  (pause-last mini))

=> 1
```

SYNOPSIS:

```
(defmethod pause-last ((sc slippery-chicken)
                        &key
                          ;; MDE Tue Feb 26 09:07:43 2019 -- type of pause added
                          ;; can vary
                          (pause 'pause)
                          ;; MDE Sat Feb 23 12:52:15 2019 -- making players a key
                          ;; arg instead of required
                          players
                          ;; we might want the pause mark somehere else
                          (bar-num (num-bars sc))
                          bar-line)
```

19.55   slippery-chicken-edit/process-events-by-time

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Apply the given function to all event objects within the given measure
 range in order of their chronological occurrence. The function can take
 one argument only: the current event object. NB If the time of the event
 is needed it can be accessed in the given function via the event's
 start-time slot.

ARGUMENTS:

 - A slippery-chicken object.
 - A function (or variable to which a function has been assigned).

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :start-bar. An integer that is the first bar in which the function is to
  be applied to event objects. Default = 1.
- :end-bar. NIL or an integer that is the last bar in which the function is
  to be applied to event objects. If NIL, the last bar of the
  slippery-chicken object is used. Default = NIL.
```

RETURN VALUE:

```
 T
```

EXAMPLE:

```
(let ((marks (make-cscl '(a s as te ts at))))
  (defun add-random-marks (event)
    (unless (is-rest event)
      (setf (marks event) (list (get-next marks))))))

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                    (va (viola :midi-channel 2))
                    (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((3 4) s (e.)  (s) s (e) (e) s (s)))
                                :pitch-seq-palette ((1 2 3))))
                            (2 ((((3 4) (s) s (e) (e) s (s) s (e.)))
                                :pitch-seq-palette ((1 2 3))))
                            (3 ((((3 4) (e) s (s) s (e.)  (s) s (e)))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 2 3))
                            (va (2 3 1))
                            (vc (3 1 2)))))))))
  (process-events-by-time mini #'add-random-marks))
```

SYNOPSIS:

```
(defmethod process-events-by-time ((sc slippery-chicken) function
                                   &key (start-bar 1) end-bar)
```

19.56   slippery-chicken-edit/re-bar

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Arrange the events of specified consecutive bars in a given
slippery-chicken object into new bars of a different time signature. If the
number of beats in the specified series of events does not fit evenly into
full measures of the the specified time signature, the method will do its
best to create occasional bars of a different time-signature that are as
close as possible to the desired length.

This method will only combine existing short bars into longer ones; it
won't split up longer bars and recombine them.

NB: This method should not be confused with the rebar method (which is used
internally, not directly by the user).

NB: as this reorganises fundamental structures within the slippery-chicken
object, it may cause methods like clm-play to fail with some arguments
which seem reasonable to the user. So use immediately before creating the
score and after creating other outputs.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments
 - :start-bar. An integer that is the number of the first bar whose events
   are to be re-barred.
 - :end-bar. An integer that is the number of the last bar whose events are
   to be re-barred.
 - :min-time-sig. A time signature in the form of a 2-item list containing
   the number of beats and the beat unit; e.g. '(3 4). This is a target time
   signature from which the method may occasionally deviate if the number of
   events does not fit evenly into full bars of the specified time
   signature.
 - :verbose. T or NIL to indicate whether to print feedback on the
   re-barring process to the Listener. T = print feedback. Default = NIL.
 - :check-ties. T or NIL to indicate whether to force the method to ensure
   that all ties have a beginning and ending. T = check.
   Default = T.
 - :auto-beam. T, NIL, or an integer. If T, the method will automatically
   attach beam indications to the corresponding events according to the beat
   unit of the time signature. If an integer, the method will beam in
   accordance with a beat unit that is equal to that integer. If NIL, the

```
    method will not automatically place beams. Default = T.
 - :update-slots. T or NIL to indicate whether to update all slots of the
   given slippery-chicken object after applying the method. This is an
   internal argument and will generally not be needed by the user.
```

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                               :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1 1)))))))))
  (re-bar mini :start-bar 2 :end-bar 5 :min-time-sig '(4 4) :auto-beam 4))

  => T
```

SYNOPSIS:

```
(defmethod re-bar ((sc slippery-chicken)
                   &key start-bar
                   end-bar
                   ;; the following is just a list like '(3 8) '(5 8)
                   min-time-sig
                   verbose
                   ;; MDE Thu Feb  9 10:36:02 2012 -- seems if we don't
                   ;; update-slots then the new bar structure isn't displayed
                   (update-slots t)
                   (check-ties t)
                   ;; could also be a beat rhythmic unit
                   (auto-beam t))
```

19.57   slippery-chicken-edit/rehearsal-letters-at-sections

[slippery-chicken-edit] [Methods]

DATE:

August 19th 2016, Edinburgh

DESCRIPTION:

Place a rehearsal letter at the beginning of every section (and subsection)
in the piece.

ARGUMENTS:

- a slippery chicken object

RETURN VALUE:

A list of the bar numbers where rehearsal letters will appear.

SYNOPSIS:

```
(defmethod rehearsal-letters-at-sections ((sc slippery-chicken))
```

19.58   slippery-chicken-edit/remove-extraneous-dynamics

[slippery-chicken-edit] [Methods]

DESCRIPTION:

A post-generation editing method: If two or more consecutive event objects
have the same dynamic, remove that dynamic marking from all but the first
of these.

ARGUMENTS:

- A slippery-chicken object.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
```

```
      :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
      :set-map '((1 (1 1 1 1 1 1 1)))
      :rthm-seq-palette '((1 ((((2 4) q e s s))
                                 :pitch-seq-palette ((1 2 3 4))
                                 :marks (f 1 f 2 f 3 f 4))))
      :rthm-seq-map '((1 ((vn (1 1 1 1 1 1 1)))))))))))
 (remove-extraneous-dynamics mini))

 => T
```

SYNOPSIS:

```
(defmethod remove-extraneous-dynamics ((sc slippery-chicken))
```

19.59   slippery-chicken-edit/replace-events

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Replace one or more consecutive existing event objects with new event
objects. All references are 1-based. This method can be applied to only one
bar at a time.

One or more new event objects can be specified as a replacement for one
single original event object.

It's the user's responsibility to make sure the bar is full and
update-slots is called after all post-generation editing.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player whose part is to be modified.
- An integer that is the number of the bar in which the change is to be
  made; or a reference to the bar in the format '(section sequence bar).
- An integer that is the number of the first event object in the given bar
  to replace.
- An integer that is the total number of consecutive original event objects
  to replace.
- A list of the new event objects, each in turn specified as a 2-item list
  in the format (pitch rhythm), e.g. '((c4 e)). Rests are indicated with
  NIL or 'r, e.g. (nil s) (r h). Chords are indicated by enclosing the
  pitches of the chord in a list, e.g. ((c4 e4) e).

OPTIONAL ARGUMENTS:

  - T or NIL to indicate whether to automatically re-beam the given bar after
    replacing the events. T = beam. Default = NIL.
  - A list of integers to indicate tuplet bracket placement, in the format
    '(tuplet-value start-event end-event). These numbers are 0-based and
    inclusive and count rests.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c2 e2 d4 e4 f4 g4 a4 f5))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (replace-events mini 'vn 1 2 1 '((nil s) ((ds5 fs5) s)) t)
  (replace-events mini 'vn 2 2 1 '((cs5 e)))
  (replace-events mini 'vn '(1 3 1) 3 1 '((df4 s)))
  (replace-events mini 'vn 4 1 1 '((ds4 te) (r te) (b3 te)) t '(3 0 2)))
```

=> T

SYNOPSIS:

```
(defmethod replace-events ((sc slippery-chicken) player bar-num start-event
                           replace-num-events new-events
                           &optional (auto-beam nil) tuplet-brackets)
```

19.60   slippery-chicken-edit/replace-multi-bar-events

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Replace specified consecutive event objects across several bars.

 The new rhythms provided must produce full bars for all bars specified;
 i.e., if only a quarter note is provided as the new event for a 2/4 bar,
 the method will not automatically fill up the remainder of the bar.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player whose part is to be modified.
- An integer that is the number of the first bar in which event objects are
  to be replaced. This can be an absolute bar number or a list in the form
  '(section sequence bar); or with subsections then e.g. '((3 1) 4 2)).
- An integer that is the number of bars in which event objects will be
  replaced.
- The list of new event objects. The new event objects can be passed as
  complete event objects; as a list of 2-item lists that are
  note-name/rhythm pairs, e.g: '((c4 q) (d4 e)); or as a list with two
  sub-lists, the first being just the sequence of rhythms and the second
  being just the sequence of pitches, e.g: '((q e) (c4 d4)). For the
  latter, :interleaved must be set to NIL. (see :interleaved below). Pitch
  data is the usual cs4 or (cs4 cd3) for chords, and NIL or 'r indicate a
  rest. NB: All pitches are sounding pitches; written pitches will be
  created for transposing instruments where necessary.

OPTIONAL ARGUMENTS:

 keyword arguments:
- :interleaved. T or NIL to indicate whether the new event data is to be
  processed as a list of note-name/rhythm pairs (or existing event
  objects), or if it is to be processed as a list with two sub-lists, the
  first containing the sequence of rhythms and the second containing the
  sequence of pitches (see above). T = interleaved, i.e. already existing
  event objects or a list of note-name/rhythm pairs. NIL = separate lists
  for rhythms and pitches. Default = T.
  If this argument is T, the list of 2-element lists (note-name/rhythm
  pairs) is passed to make-events, but such a list can contain no ties. If
  the argument is set to NIL, the rhythm and pitch data is passed as two
  separate lists to make-events2 where + can be used to indicate ties.
- :consolidate-rests. T or NIL to indicate whether shorter rests should
  automatically be consolidated into a single longer rest.
  T = consolidate. Default = T.
  NB: slippery chicken will always consolidate full bars of rest into
  measure-rests, regardless of the value of this argument.
- :beat. NIL or an integer (rhythm symbol) that indicates which beat basis
  will be used when consolidating rests. If NIL, the beat of the time
  signature will be used (e.g. quarter in 4/4). Default = NIL.
- :auto-beam. T or NIL to indicate whether to automatically beam the new
  events. T = automatically beam. Default = T.
- :tuplet-bracket. NIL or an integer to indicate whether to automatically
  add tuplet (e.g. triplet/quintuplet) brackets to the new events where
  applicable. If this is an integer, all tuplets in the given bar will be

given a tuplet bracket with that integer as the tuplet number. NB: This
option does not allow for setting tuplets of different numbers for the
same bar. To do that, set :tuplet-bracket to NIL and add the
tuplet-brackets manually. NIL = place no brackets. Default = NIL.

RETURN VALUE:

The number of new events used to replace the old ones.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((d4 e4 f4 g4))))
        :set-map '((1 (1 1 1 1 1 1))
                   (2 (1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))))
                            (2 ((((2 4) e s s q)
                                 (s s e +e e))
                                :pitch-seq-palette ((1 2 3 4 3 2 4 1)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1))))
                        (2 ((vn (2 2 2 2 2 2)))))))))
  (replace-multi-bar-events mini 'vn 2 3
                            '((cs5 h) ((ds5 fs5) h) (nil h)))
  (replace-multi-bar-events mini 'vn '(2 2 2) '3
                            '((h h h) (cs5 (ds5 fs5) nil))
                            :interleaved nil)
  (replace-multi-bar-events mini 'vn 1 1
                            '((nil e) (nil e) (nil e) (cs4 e))
                            :consolidate-rests t)
  (replace-multi-bar-events mini 'vn 8 1
                            '((nil q) (b3 e) (cs4 s) (ds4 s))
                            :auto-beam t))

=> 4
```

SYNOPSIS:

```
(defmethod replace-multi-bar-events ((sc slippery-chicken)
                                     player start-bar num-bars new-events
                                     &key
                                        ;; 24.3.11: see above.
```

```
                                          (interleaved t)
                                          ;; MDE Mon Apr 23 12:36:08 2012 --
                                          ;; changed default to nil
                                          (consolidate-rests nil)
                                          ;; for consolidate rests
                                          (beat nil)
                                          ;; MDE Mon Apr 23 12:36:08 2012 --
                                          ;; changed default to nil
                                          (auto-beam nil)
                                          ;; MDE Fri Aug 29 10:18:29 2014
                                          (warn t)
                                          ;; MDE Mon Sep  1 16:41:02 2014
                                          (delete-beams t)
                                          (delete-tuplets t)
                                          ;; MDE Sat Jan 17 17:52:07 2015 --
                                          ;; should we update the player slot of
                                          ;; each event?
                                          (write-player nil)
                                          ;; 31.3.11: if this is t, then rthms > a
                                          ;; beat will case an error
                                          (auto-beam-check-dur t)
                                          (tuplet-bracket nil))
```

19.61   slippery-chicken-edit/replace-tempo-map

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Replace the tempo data for a given slippery-chicken object with new
specified tempo indications.

Calls not only the setf method - which converts bar references like
(section-num sequence-num bar-num) to numbers and makes a tempo-map object,
but also updates all event objects to reflect new start times etc.

ARGUMENTS:

- A slippery-chicken object
- A list that is the new tempo-map.

RETURN VALUE:

T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((pno (piano :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 f4 g4 a4 c5 d5 f5 g5 a5 c6))))
        :set-map '((1 (1 1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q q))
                                :pitch-seq-palette ((1 (2))))))
        :rthm-seq-map '((1 ((pno (1 1 1 1 1 1 1 1)))))))))
  (replace-tempo-map mini '((1 (q 60 "Andante")) ((1 3 1) (e 80)))))
```

=> T

SYNOPSIS:

```
(defmethod replace-tempo-map ((sc slippery-chicken) tm)
```

19.62   slippery-chicken-edit/reset-midi-channels

[slippery-chicken-edit] [Methods]

DATE:

 June 9th 2020

DESCRIPTION:

 Reset players' event objects to use the midi-channels given in the ensemble
 rather than those inherited created elsewhere.

ARGUMENTS:

 - the slippery-chicken object to process

OPTIONAL ARGUMENTS:

 - a symbol or list of symbols representing the players we want to process
   (i.e. the same player symbols found in the ensemble slot, not
   instruments). Default = NIL = all players
 - the start-bar in which events should begin to be processed. Default = 1
 - the end bar (inclusive) in which the last events should be
   processed. Default = NIL = the last bar of the slippery-chicken object.

RETURN VALUE:

 T

SYNOPSIS:

19.63   slippery-chicken-edit/respell-bars

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Look for enharmonically equivalent pitches in the same bar and try to unify
 their spelling. The method applies this process to every bar in the given
 slippery-chicken object.

 Also see rthm-seq-bar/respell-bar and slippery-chicken/respell-notes.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 df5 ef5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))))))))
  (respell-bars mini))

=> T
```

SYNOPSIS:

```
(defmethod respell-bars ((sc slippery-chicken))
```

19.64   slippery-chicken-edit/respell-notes

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Pass through the entire given slippery-chicken object and change some of
the pitch objects to their enharmonic equivalents to produce more sensible
spellings of consecutive pitches in the score.

An optional argument takes a list specifying which pitches to change in the
same format found in the method enharmonic-spellings; i.e.
'((player (bar note-num))). These notes are changed after the respelling
routine has run.

NB: If a list of corrections is specified, the :respell-notes argument of
    any subsequent call to cmn-display or write-lp-data-for-all must be set
    NIL, otherwise the modified pitches may be overwritten. Also, although
    this algorithm corrects tied notes when respelling, notes referenced in
    the corrections list will not be followed through to any subsequent
    ties.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

 - A list of specific notes whose pitches are to be enharmonically flipped,
   in the format, e.g. '((vn (1 1) (1 4)) (vc (2 3) (3 3)))

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;; An example using respell-notes for the whole slippery-chicken object.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 df5 ef5))))
        :set-map '((1 (1 1 1 1 1)))
```

```
         :rthm-seq-palette '((1 ((((2 4) q e s s))
                                  :pitch-seq-palette ((1 2 3 4)))))
         :rthm-seq-map '((1 ((vn (1 1 1 1 1))))))))
  (respell-notes mini))

;; An example specifying which pitches are to be enharmonically changed.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 df5 ef5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                  :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))))))))
  (respell-notes mini '((vn (1 1) (1 4))))
  (cmn-display mini :respell-notes nil))

=> T
```

SYNOPSIS:

```
(defmethod respell-notes ((sc slippery-chicken) &optional corrections)
```

19.65   slippery-chicken-edit/respell-notes-for-player

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Pass through the pitches of a specified player's part and change some of
the pitches to their enharmonic equivalents in order to produce more
sensible spellings of consecutive notes.

This is just a very simple attempt to better spell notes by comparing each
note to the previous two and making it the same accidental type. It
doesn't look further back or ahead as of yet.

If the optional argument is set to T, then look at the written notes
instead of the sounding notes.

NB: Since both the cmn-display and write-lp-data-for-all methods
    automatically call respell-notes for all players of an entire
    sc-object, their :respell-notes argument may need to be set to NIL for
    this method to produce the desired results.

ARGUMENTS:

 - A slippery-chicken object.
 - The ID of the player whose pitches are to be modified.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to change written pitches only or sounding
   pitches only. T = change written pitches only. Default = NIL.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vn (violin :midi-channel 2))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((b3 cs4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vn (1 1 1 1 1)))))))))
  (respell-notes-for-player mini 'cl t)
  (cmn-display mini :respell-notes nil :in-c nil))

=> T
```

SYNOPSIS:

```
(defmethod respell-notes-for-player ((sc slippery-chicken) player
                                      &optional written)
```

19.66   slippery-chicken-edit/rest-to-note

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Change a specified event object from a rest into a note by supplying a
pitch or chord (as objects or symbols).

Marks to be attached to the new note can be supplied as a symbol or a list
as an optional argument.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the rest is to be
  changed to a note.
- An integer that is the number of the rest in the given bar that is to be
  changed. This number counts rests only, not sounding notes or events.
- The ID of the player whose part is to be changed.
- A note-name symbol that is to be the pitch of the new note, or a list of
  note-name symbols that will make up a chord.

OPTIONAL ARGUMENTS:

- A mark or list of marks to be attached to the new note.

RETURN VALUE:

Returns the new event object created.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 fs4))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                 :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1)))))))))
  (rest-to-note mini 2 1 'vn 'gs5)
  (rest-to-note mini 3 1 'vn '(gs5 b5))
  (rest-to-note mini 4 1 'vn '(gs4 b4) 'ppp)
  (rest-to-note mini 5 1 'vn '(gs4 b4) '(fff pizz)))

=>
EVENT: start-time: 9.000, end-time: 9.500,
       duration-in-tempo: 0.500,
```

```
        compound-duration-in-tempo: 0.500,
        amplitude: 0.900
        bar-num: 5, marks-before: NIL,
        tempo-change: NIL
        instrument-change: NIL
        display-tempo: NIL, start-time-qtrs: 9.000,
        midi-time-sig: NIL, midi-program-changes: NIL,
        8va: 0
        pitch-or-chord:
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
[...]
```

SYNOPSIS:

```
(defmethod rest-to-note ((sc slippery-chicken) bar-num rest-num player new-note
                         &rest marks)
```

19.67   slippery-chicken-edit/rm-marks-from-note

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Remove one or more specific marks from the MARKS slot of a specified event
object.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar from which the marks are to be
  removed.
- An integer that is the number of the note in that bar from which the
  marks are to be removed.
- The ID of the player from whose part the marks are to be removed.

OPTIONAL ARGUMENTS:

- A specific mark or list of specific marks that are to be removed. If this
  argument is not specified, no marks will be removed.

RETURN VALUE:

```
 Returns T.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 fs4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                 :pitch-seq-palette ((1 2 3 4))
                                 :marks (a 2 s 2 fff 2 pizz 2))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (rm-marks-from-note mini 2 2 'vn 'pizz)
  (rm-marks-from-note mini 3 2 'vn '(pizz fff))
  (rm-marks-from-note mini 3 2 'vn))

=> T
```

SYNOPSIS:

```
(defmethod rm-marks-from-note ((sc slippery-chicken) bar-num note-num
                                player &rest marks)
```

19.68   slippery-chicken-edit/rm-marks-from-notes

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Remove only the specified marks from the MARKS slots of specified events in
the parts of specified players. If the <players> argument is set to NIL,
remove the mark or marks from all players.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer or a 2-item list of integers indicating the first bar and note
   from which to remove marks. If an integer, this is the bar number and the
   mark will be removed from all notes in the bar. If a 2-item list, this is
   a reference to the bar number and number of the first note in the bar
   from which to start removing marks, in the form e.g. '(3 1).
 - An integer or a 2-item list of integers indicating the last bar and note

from which to remove marks. If an integer, this is the bar number and the
mark will be removed from all notes in the bar. If this is a 2-item list,
this is a reference to the bar number and number of the first note in the
bar from which to start removing marks, in the form e.g. '(3 1).
- The ID or a list of IDs of the players from whose parts the marks are to
  be removed.

OPTIONAL ARGUMENTS:

NB: The <marks> argument is a required argument for this method.
- The mark or a list of the marks to remove. This method will only remove
  specified marks.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((fl (flute :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vn (violin :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((cs4 ds4 fs4))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))
                                :marks (a 2 s 2 fff 2))))
        :rthm-seq-map '((1 ((fl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vn (1 1 1 1 1)))))))))
  (rm-marks-from-notes mini 1 2 'fl 'fff)
  (rm-marks-from-notes mini '(1 2) '(2 1) 'hn '(fff a))
  (rm-marks-from-notes mini 3 '(4 3) '(hn vn) '(fff s a))
  (rm-marks-from-notes mini 5 5 nil 'fff))

=> T
```

SYNOPSIS:

```
(defmethod rm-marks-from-notes ((sc slippery-chicken) start end
                                      players &rest marks)
```

19.69   slippery-chicken-edit/rm-pitches-from-chord

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Remove the specified pitches from an existing chord object.

ARGUMENTS:

 - The slippery-chicken object which contains the given chord object.
 - The ID of the player whose part is to be affected.
 - An integer that is the number of the bar that contains the chord object
   that is to be modified.
 - An integer that is the number of the note that is the chord object to be
   modified.
 - The pitches to be removed. These can be pitch objects or any data that
   can be passed to make-pitch, or indeed lists of these, as they will be
   flattened.

RETURN VALUE:

 The chord object that has been changed.

EXAMPLE:

```
(let* ((ip-clone (clone +slippery-chicken-standard-instrument-palette+)))
  (set-slot 'chord-function 'chord-fun2 'guitar ip-clone)
  (let* ((mini
          (make-slippery-chicken
           '+mini+
           :instrument-palette ip-clone
           :ensemble '(((gtr (guitar :midi-channel 1))))
           :set-palette '((1 ((e3 f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5 d5 e5 f5
                               g5 a5 b5 c6 d6 e6))))
           :set-map '((1 (1)))
           :rthm-seq-palette
           '((1 ((((4 4) e e e e e e e e))
                 :pitch-seq-palette ((1 (2) 3 (4) 5 (6) 7 (8))))))
           :rthm-seq-map '((1 ((gtr (1)))))))))
    (print (get-pitch-symbols
            (pitch-or-chord (get-event mini 1 2 'gtr))))
    (rm-pitches-from-chord mini 'gtr 1 2 'a3 'd4)
    (print (get-pitch-symbols
            (pitch-or-chord (get-event mini 1 2 'gtr))))))
```

```
=>
(E3 A3 D4 G4)
(E3 G4)
```

SYNOPSIS:

```
(defmethod rm-pitches-from-chord ((sc slippery-chicken) player bar-num
                                  note-num &rest pitches)
```

19.70   slippery-chicken-edit/rm-repeated-pitches

[slippery-chicken-edit] [Methods]

DATE:

 April 23rd 2016, Edinburgh

DESCRIPTION:

 Sometimes, even pitch curves without repeated notes result in repeated
 notes in the generated piece. Often it's the removal of large fast leaps in
 instrumental parts that ends up creating repeated notes. This method will
 do its best to replace repeated notes, even in chords, with other notes
 from the current set and within the range of the current instrument and any
 set-limits. If a chord is required, then the current instrument's chord
 function will be used to create the replacement chord.

 NB If you're interested in instrument/players statistics like tessitura
 you'll have to call update-slots after you're done calling this and related
 methods.

ARGUMENTS:

 - the slippery-chicken object
 - the player (symbol)

OPTIONAL ARGUMENTS:

 - the start bar (integer). Default = 1.
 - the end bar (integer). Default = NIL which means we'll process up to the
   end of the piece.

RETURN VALUE:

 The slippery-chicken object with all possible repeated notes removed.

SYNOPSIS:

```
(defmethod rm-repeated-pitches ((sc slippery-chicken) player
                                &optional (start-bar 1) end-bar)
```

19.71   slippery-chicken-edit/rm-slurs

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Remove the specified slurs from the MARKS slots of specified events in
the parts of specified players. If the <players> argument is set to NIL,
remove the specified slurs from all players.

ARGUMENTS:

- A slippery-chicken object.
- An integer or a 2-item list of integers indicating the first bar and note
  from which to remove slurs. If an integer, this is the bar number and the
  slurs will be removed from all notes in the bar. If a 2-item list, this is
  a reference to the bar number and number of the first note in the bar
  from which to start removing slurs, in the form e.g. '(3 1).
- An integer or a 2-item list of integers indicating the last bar and note
  from which to remove slurs. If an integer, this is the bar number and the
  slurs will be removed from all notes in the bar. If this is a 2-item list,
  this is a reference to the bar number and number of the first note in the
  bar from which to start removing slurs, in the form e.g. '(3 1).
- The ID or a list of IDs of the players from whose parts the marks are to
  be removed.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((fl (flute :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vn (violin :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 e4 fs4 gs4 as4 c5 d5))))
```

```
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                              :pitch-seq-palette ((1 2 3 4 5 6 7 8))
                              :marks (slur 1 2 slur 3 4 slur 5 6 slur 7 8))))
        :rthm-seq-map '((1 ((fl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vn (1 1 1 1 1)))))))))
  (rm-slurs mini 1 2 'fl)
  (rm-slurs mini '(1 3) '(2 1) 'hn)
  (rm-slurs mini 3 '(4 3) '(hn vn))
  (rm-slurs mini 5 5 nil))

=> T
```

SYNOPSIS:

```
(defmethod rm-slurs ((sc slippery-chicken) start end players)
```

19.72   slippery-chicken-edit/round-to-nearest

[slippery-chicken-edit] [Methods]

DATE:

 October 18th 2018, Heidhausen

DESCRIPTION:

 For the given bars and players, round the events' pitch data to the nearest
 in the current or given scale. For example, a slippery-chicken object could
 be generated in the :quarter-tone scale, then the scale changed to
 :chromatic (or the keyword argument :scale 'chromatic-scale given), this
 method then called, and all pitches would be rounded to the nearest
 chromatic pitch. See the pitch class method for more details.

ARGUMENTS:

 the slippery-chicken argument

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start-bar. A number that is the first bar which should be
   processed. Default = NIL in which case 1 will be used.
 - :end-bar. A number that is the last bar which should be processed. Default

```
    = NIL in which case all bars will be processed.
 - :players. A list of the IDs of the players to whose parts the function
   should be applied. Can also be a single symbol. If NIL then all players
   will be processed.
 - :scale. The scale to use when rounding. (Common Music tuning object or
   symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or
   'quarter-tone only at present. Default is the current scale as set by
   (in-scale :...).
```

RETURN VALUE:

 the modified slippery-chicken object

SYNOPSIS:

```
(defmethod round-to-nearest ((sc slippery-chicken)
                             &key start-bar end-bar players (scale cm::*scale*))
```

19.73   slippery-chicken-edit/sc-delete-beams

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Delete beam indications from specified notes. If only a bar number is
 specified, this method deletes all beams in the bar.

```
 NB: If specifying start and end notes, the start notes specified must be
     the first note of a beamed group of notes (i.e. the BEAMS slot of the
     corresponding event object must be 1), and the end note must be the
     last note of a beamed group of notes (i.e., the BEAMS slot of the
     corresponding event object must be 0), otherwise errors may
     occur. Also, if specifying one of these arguments, both must be
     specified.
```

ARGUMENTS:

```
 - A slippery-chicken object.
 - An integer that is the number of the bar in which the beams are to be
   deleted.
 - The ID of the player from whose part the beams are to be deleted.
```

OPTIONAL ARGUMENTS:

```
 - An integer that is the number of the note that currently holds the
```

start-beam information (i.e., the BEAMS slot is 1). This number is
1-based and counts ties.
- An integer that is the number of the note that currently holds the
  end-beam information (i.e., the BEAMS slot is 0). This number is 1-based
  and counts ties.

RETURN VALUE:

 If deleting all beams in a bar, returns T, otherwise returns NIL.

EXAMPLE:

```
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vc (cello :midi-channel 1))))
         :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
         :set-map '((1 (1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) - e e - - e e - - e e - - e e -))
                                 :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
         :rthm-seq-map '((1 ((vc (1 1 1 1)))))))))
  (sc-delete-beams mini 2 'vc)
  (sc-delete-beams mini 3 'vc 3 4))

  => NIL
```

SYNOPSIS:

```
(defmethod sc-delete-beams ((sc slippery-chicken) bar-num player
                            &optional start-note end-note)
```

19.74   slippery-chicken-edit/sc-delete-marks

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Delete all marks from the MARKS slot of a given note event object and
 set the slot to NIL.

 NB: This method counts notes, not rests, and is 1-based.

ARGUMENTS:

 - A slippery-chicken object.

- An integer that is the number of the bar in which the marks are to be
  deleted.
- An integer that is the number of the note from which the marks are to be
  deleted.
- The ID of the player from whose part the marks are to be deleted.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((cs4 ds4 fs4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s s))
                                :pitch-seq-palette ((1 2 3))
                                :marks (a 2 s 2 fff 2 pizz 2))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (sc-delete-marks mini 2 2 'vn))

=> T
```

SYNOPSIS:

```
(defmethod sc-delete-marks ((sc slippery-chicken) bar-num note-num player)
```

19.75   slippery-chicken-edit/sc-delete-marks-before

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Deletes all data from the MARKS-BEFORE slot of a specified event object and
 replaces it with NIL.

 NB: In addition to clef symbol data, the MARKS-BEFORE slot also stores part
     of the required data for trills and arrows. Deleting just the
     MARKS-BEFORE components of those markings may result in unwanted
     results.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the event object is to
  be modified.
- An integer that is the number of the note within the given bar for which
  the MARKS-BEFORE slot is to be set to NIL.
- The ID of the player whose part is to be affected.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (add-mark-before-note mini 2 3 'vc 'fff)
  (add-mark-before-note mini 2 3 'vc 's)
  (add-mark-before-note mini 2 3 'vc 'lhp)
  (sc-delete-marks-before mini 2 3 'vc))

=> NIL
```

SYNOPSIS:

```
(defmethod sc-delete-marks-before ((sc slippery-chicken)
                                                bar-num note-num player)
```

19.76   slippery-chicken-edit/sc-delete-marks-from-event

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Delete all data from the MARKS slot of the specified event object and
 replace it with NIL.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar from which the marks are to be deleted.
- An integer that is the number of the event within the given bar from which the marks are to be deleted.
- The ID of the player from whose part the marks are to be deleted.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))
                                :marks (a 1 4 lhp 4 s 3 4 slur 1 2))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (sc-delete-marks-from-event mini 2 4 'vc))
```

=> NIL

SYNOPSIS:

```
(defmethod sc-delete-marks-from-event ((sc slippery-chicken)
                                        bar-num event-num player)
```

19.77   slippery-chicken-edit/sc-force-rest

[slippery-chicken-edit] [Methods]
DATE:

 23-Jul-2011 (Pula)

DESCRIPTION:

 Change the specified event object to a rest. If events tied from this
 event should automatically be forced to rests also, use the sc-force-rest2
 method instead.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the rest is to be
  forced.
- An integer that is the number of the event within that bar which is to be
  changed into a rest. This number is 1-based and counts tied notes but not
  rests.
- The ID of the player whose part is to be modified.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the specified bar should be automatically
  beamed after the change has been made. NB: In general, calling auto-beam
  is a good idea (esp. when deleting notes under an existing beam);
  however, auto-beam may fail when addressing bars that contain notes
  longer than one beat. T = automatically beam. Default = NIL.

RETURN VALUE:

 The new rthm-seq-bar object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (sc-force-rest mini 2 3 'vc)
  (sc-force-rest mini 3 3 'vc t))

  =>

  RTHM-SEQ-BAR: time-sig: 3 (2 4), time-sig-given: T, bar-num: 3,
  old-bar-nums: NIL, write-bar-num: NIL, start-time: 4.000,
  start-time-qtrs: 4.0, is-rest-bar: NIL, multi-bar-rest: NIL,
  show-rest: T, notes-needed: 3,
  tuplets: NIL, nudge-factor: 0.35, beams: ((1 2)),
  current-time-sig: 3, write-time-sig: NIL, num-rests: 1,
  num-rhythms: 4, num-score-notes: 3, parent-start-end: NIL,
```

```
 missing-duration: NIL, bar-line-type: 2,
 player-section-ref: (1 VC), nth-seq: 2, nth-bar: 0,
 rehearsal-letter: NIL, all-time-sigs: (too long to print)
 sounding-duration: 1.750,
 rhythms: (
[...]
```

SYNOPSIS:

```
(defmethod sc-force-rest ((sc slippery-chicken) bar-num note-num player
                          &optional auto-beam)
```

19.78   slippery-chicken-edit/sc-force-rest2

[slippery-chicken-edit] [Methods]

DESCRIPTION:

```
Turn events into rests, doing the same with any following tied events.
NB As it is foreseen that this method may be called many times iteratively,
there is no call to check-ties, auto-beam, consolidate-rests, or
update-instrument-slots (for statistics)--it is advised that these methods
are called once the last call to this method has been made.  gen-stats is
however called for each affected bar, so the number of rests vs. notes
should be consistent with the new data.
```

```
NB This fill trigger an error if attempted with grace-notes
```

ARGUMENTS:

```
 - A slippery-chicken object
 - The bar number (integer)
 - The event number in the bar (integer, counting from 1)
 - The player name (symbol)
```

OPTIONAL ARGUMENTS:

```
 - A function object to be called on error (could be #'error (default),
 #'warn, #'print or simply NIL for no error)
```

RETURN VALUE:

```
 The number of events turned into rests.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette
        '((1 ((((4 4) - e.. 32 - h.) (+w) (+w) ((w)) ((h) (e) q e)
               (+q - +s e. - +h) (+w) (+w) ((w))))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (sc-force-rest2 mini 1 3 'vc))
=> 3
```

SYNOPSIS:

```
(defmethod sc-force-rest2 ((sc slippery-chicken) bar-num event-num player
                            &optional (on-error #'error))
```

19.79   slippery-chicken-edit/sc-move-dynamic

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Move the dynamic attached to a specified event object to another specified
event object.

By default the dynamics are moved between events within the same bar. An
optional argument allows for dynamics to be moved to events in a different
bar.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which to move the dynamic.
- The ID of the player in whose part the dynamic is located.
- An integer that is the number of the event object from which the dynamic
  is to be moved. This number is 1-based and counts both rests and ties.
- An integer that is the number of the event object to which the dynamic
  is to be moved. This number is 1-based and counts both rests and ties.

OPTIONAL ARGUMENTS:

- An integer that is the number of the bar to which the dynamic should be
  moved. If this is not specified, the dynamic will be moved to the
  specified event within the same bar.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))
                                :marks (fff 1))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (sc-move-dynamic mini 1 'vc 1 3)
  (sc-move-dynamic mini 2 'vc 1 4 3))

  => T
```

SYNOPSIS:

```
(defmethod sc-move-dynamic ((sc slippery-chicken) bar-num player
                            ;; event numbers 1-based but counting rests and ties
                            from to &optional to-bar)
```

19.80   slippery-chicken-edit/sc-remove-dynamic

[slippery-chicken-edit] [Methods]

DESCRIPTION:

 Remove all dynamics from the MARKS slot of one or more specified event
 objects.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar from which the dynamics are to
   be removed.
 - The ID of the player from whose part the dynamics are to be removed.
 - An integer or a list of integers that are the numbers of the events from
   which the dynamics are to be removed. Event numbers include ties and
   rests.

RETURN VALUE:

 Returns the last dynamic removed.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((a3 b3 c4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))
                                :marks (fff 1 ppp 3))))
        :rthm-seq-map '((1 ((vc (1 1 1)))))))))
  (sc-remove-dynamic mini 2 'vc 1)
  (sc-remove-dynamic mini 3 'vc '(1 3)))

  => PPP
```

SYNOPSIS:

```
(defmethod sc-remove-dynamic ((sc slippery-chicken) bar-num player
                              &rest event-nums)
```

19.81   slippery-chicken-edit/sc-remove-dynamics

[slippery-chicken-edit] [Methods]

DATE:

 16-Mar-2011

DESCRIPTION:

 Remove all dynamic marks from the MARKS slots of all consecutive event
 objects (or just rests, if preferred) within a specified region of bars.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer or a list of two integers. If a single integer, this is the
   number of the first bar from which the dynamics will be removed, and all

dynamics will be removed from the full bar. If this is a list of two
integers, they are the numbers of the first bar and first note within
that bar from which the dynamics will be removed, in the form '(bar-num
note-num). Note numbers are 1-based and count ties but not rests.
- An integer or a list of two integers. If a single integer, this is the
number of the last bar from which the dynamics will be removed, and all
dynamics will be removed from the full bar. If this is a list of two
integers, they are the numbers of the last bar and last note within that
bar from which the dynamics will be removed, in the form '(bar-num
note-num). Note numbers are 1-based and count ties but not rests.
- A single ID or a list of IDs of the players from whose parts the dynamics
are to be removed. If NIL then all players will be processed. Default =
NIL = all players.

OPTIONAL ARGUMENTS:

- T or NIL to indicate that dynamics should be removed only from
rests. Default = NIL.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((d3 e3 f3 g3 a3 b3 c4 e4 f4 g4 a4 b4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4))
                                :marks (fff 1 ppp 3))))
        :rthm-seq-map '((1 ((vn (1 1 1))
                            (va (1 1 1))
                            (vc (1 1 1)))))))))
  (sc-remove-dynamics mini '(1 2) '(2 2) 'vn)
  (sc-remove-dynamics mini 2 3 '(va vc)))

 => T
```

SYNOPSIS:

```
(defmethod sc-remove-dynamics ((sc slippery-chicken) start end players
                               &optional just-rests)
```

19.82   slippery-chicken-edit/set-cautionary-accidental

[slippery-chicken-edit] [Methods]

DATE:

 28-Sep-2011

DESCRIPTION:

 A post-generation editing method: Place a cautionary accidental
 (sharp/flat/natural sign in parentheses) before a specified note.

 NB: Adding cautionary accidentals to pitches within chords is currently
     only possible in LilyPond output. Adding cautionary accidentals to
     single pitches is possible in both CMN and LilyPond.

 NB: Since the cmn-display and write-lp-data-for-all methods call
     respell-notes by default, that option must be explicitly set to NIL
     within the calls to those methods in order for this method to be
     effective.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar in which to add the cautionary
   accidental.
 - An integer or a 2-item list of integers that is the number of the note
   within that bar to which to add the cautionary accidental. This number is
   1-based and counts ties. If a 2-item list such, this indicates that the
   pitch is within a chord; e.g., '(1 2) indicates that a cautionary
   accidental should be added to the 2nd pitch up from the bottom of the
   chord located at the 1st note position in the bar.
 - The ID of the player to whose part the cautionary accidental is to be
   added.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to add the cautionary accidental to only the
   written pitch or only the sounding pitch. T = written only.
   Default = NIL.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (pn (piano :midi-channel 2))))
        :set-palette '((1 ((ds3 e3 fs3 af3 bf3 c4 ef4 fs4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 (3) 4))
                                :marks (fff 1 ppp 3))))
        :rthm-seq-map '((1 ((cl (1 1 1))
                            (pn (1 1 1)))))))))
  (respell-notes mini)
  (set-cautionary-accidental mini 3 2 'cl t)
  (set-cautionary-accidental mini 2 1 'pn)
  (set-cautionary-accidental mini 2 2 'pn)
  (set-cautionary-accidental mini 3 '(3 3) 'pn)
  (write-lp-data-for-all mini :respell-notes nil))

=> T
```

SYNOPSIS:

```
(defmethod set-cautionary-accidental ((sc slippery-chicken) bar-num note-num
                                      player &optional written)
```

19.83   slippery-chicken-edit/set-limits-by-section

[slippery-chicken-edit] [Methods]

DATE:

 February 20th 2018, Heidhausen

DESCRIPTION:

 For each section/subsection of a piece, set the highest or lowest pitch,
 i.e. create the curve for one player in the set-limits-high or
 set-limits-low slot.

N.B.1 As well as returning the generated list, the respective slot value
(<which>) is also changed.

N.B.2 As the internal structure of the slippery-chicken object is modified
here, and this has implications for pitch-selection, i.e. constraints for an
already-existing slippery-chicken object (made, probably, by
make-slippery-chicken), the piece will have to be regenerated by explicitly
calling (sc-init) after calling this method.

ARGUMENTS:

- the slippery-chicken object
- the pitches: a list of pitch symbols or MIDI note numbers. Note that these
  will be used cyclically if there aren't enough pitches for the number of
  sections/subsections in the piece.
- 'set-limits-high or 'set-limits-low to set the desired curve

OPTIONAL ARGUMENTS:

- which player (symbol) to set the curve for. Default = 'all, which means
  the curve will apply to all players.

RETURN VALUE:

 The curve created (list)

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((f3 g3 a3 b3 c4))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1 1))))
                        (2 ((vc (1 1 1 1 1))))
                        (3 ((vc (1 1 1 1 1)))))))))
  (flet ((print-em ()
                   ;;                    first bars of sections 1, 2, 3
                   (loop for bar in '(1 6 11) do
```

```
                              (print (get-pitch-symbols (get-bar mini bar 'vc))))))
    (print-em)
    ;; replace existing curve. NB The y values are degrees which if default is
    ;; quarter-tone scale might be the double of what is perhaps  expected
    ;; (i.e. (note-to-degree 'c4) -> 120)
    (print (set-limits-by-section mini '(g3 a3 c4) 'set-limits-high 'vc))
    (sc-init mini)
    ;; note the new (repeating) pitches
    (print-em)))

---->>

******* section (1)
Getting notes for VC
******* section (2)
Getting notes for VC
******* section (3)
Getting notes for VC
WARNING:
   slippery-chicken::tempo-curve-to-map: No tempo-map or tempo-curve given.
Using default of crotchet/quarter = 60.
Shortening short, fast leaps...
Shortened 0 large fast leaps
(F3 G3 A3 B3 C4)
(F3 G3 A3 B3 C4)
(F3 G3 A3 B3 C4)
(1 110 5 110 6 114 10 114 11 120 15 120)
******* section (1)
Getting notes for VC
******* section (2)
Getting notes for VC
******* section (3)
Getting notes for VC
Shortening short, fast leaps...
Shortened 0 large fast leaps
(F3 F3 G3 G3 G3)
(F3 G3 G3 A3 A3)
(F3 G3 A3 B3 C4)
NIL
```

SYNOPSIS:

```
(defmethod set-limits-by-section ((sc slippery-chicken) pitches which
                                  &optional (player 'all))
```

19.84   slippery-chicken-edit/set-midi-channels

[slippery-chicken-edit] [Methods]

DATE:

 October 25th 2018, Heidhausen

DESCRIPTION:

 Change the midi-channels of events in the slippery-chicken object. All
 events' pitch objects' midi-channel slots will be destructively overwritten
 when using this method.

 The second argument is a list of lists. Each sublist has the player (not
 instrument) ID symbol as its first element, followed by one or two channel
 numbers. If only one channel is given (i.e. it's a two-element list
 e.g. (vln 1)) then it will be used for both chromatic and microtonal pitches
 (i.e. all the players' pitches will be written to the one given midi
 channel). Otherwise if a second channel is given (e.g. (vln 1 2)) then
 microtonal pitches will be written on the second channel.

ARGUMENTS:

 - the slippery-chicken object.
 - a list of player and channel info (see above and example below).

OPTIONAL ARGUMENTS:

 - the start-bar in which events should begin to be processed. Default = 1
 - the end bar (inclusive) in which the last events should be
   processed. Default = NIL = the last bar of the slippery-chicken object.

RETURN VALUE:

 T

EXAMPLE:

```
(set-midi-channels +sc-obj+ '((solo 1) (computer-a 2) (computer-b 3)
                             ;; this one uses separate midi channels for
                             ;; chromatic and microtonal pitches
                             (computer-c 4 5)))
```

SYNOPSIS:

```
(defmethod set-midi-channels ((sc slippery-chicken) channel-info
                              &optional (start-bar 1) end-bar)
```

19.85   slippery-chicken-edit/set-rehearsal-letter

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add the specified rehearsal letter/number to the specified bar in one or
more specified players.

NB: Since internally this method actually attaches the rehearsal
    letter/number to the REHEARSAL-LETTER slot of the preceding bar
    (bar-num - 1), no rehearsal letter/number can be attached to the first
    bar.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar to which the rehearsal
  letter/number is to be added.
- A symbol that is the rehearsal letter/number to be added (e.g. 'A or '1)

OPTIONAL ARGUMENTS:

- The player ID or a list of player IDs to whose parts the rehearsal
  letter/number is to be added. If no value is given here, the rehearsal
  letter/number will be added to the first (top) instrument in each group
  of the ensemble, as specified in staff-groupings.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vn (violin :midi-channel 1))
                      (va (viola :midi-channel 2))
                      (vc (cello :midi-channel 3))))
         :set-palette '((1 ((ds3 e3 fs3 af3 bf3 c4 ef4 fs4))))
         :set-map '((1 (1 1 1 1)))
         :rthm-seq-palette '((1 ((((2 4) q e s s))
                                 :pitch-seq-palette ((1 2 3 4)))))
         :rthm-seq-map '((1 ((vn (1 1 1 1))
```

```
                              (va (1 1 1 1))
                              (vc (1 1 1 1)))))))))
  (set-rehearsal-letter mini 2 'A)
  (set-rehearsal-letter mini 3 '2 '(va vc))
  (set-rehearsal-letter mini 4 'Z3))

  => T
```

SYNOPSIS:

```
(defmethod set-rehearsal-letter ((sc slippery-chicken) bar-num letter
                                 &optional players)
```

19.86 slippery-chicken-edit/swap-marks

[slippery-chicken-edit] [Methods]

DATE:

 Thu 22 Aug 2019 17:12:27 BST

DESCRIPTION:

 Replace one dynamic mark with another, or a list of marks with another
 list.

ARGUMENTS:

 - the slippery chicken object which contains marks to be swapped
 - start-bar: the first bar to start swapping
 - end-bar: the last bar to end swapping
 - players: the player or players
 - old-marks: a single mark or a list of marks
 - new-marks: a single mark to a list. NB if a list it must be the same
 length as the old-marks list. Maybe I should change this?

OPTIONAL ARGUMENTS:

 nil

RETURN VALUE:

 The number of swapped marks

EXAMPLE:

```
(let* ((mini (make-slippery-chicken
                '+mini+
                :ensemble '(((flt (flute :midi-channel 1))))
                :staff-groupings '(1)
                :tempo-map '((1 (q 60)))
                :set-palette '((set1 ((fs2 b2 d4 a4 d5 e5 a5 d6)))
                               (set2 ((b2 fs3 d4 e4 a4 d5 e5 a5 d6))))
                :set-map '((1 (set1 set1 set2 set1 set1 set2)))
                :rthm-seq-palette
                '((seq1 ((((4 4) (q) (q) q q))
                            :pitch-seq-palette (1 2)
                            :marks (pp 1)))
                  (seq2 ((((4 4) (e) e q h))
                            :pitch-seq-palette (1 2 3)
                            :marks (p 1 a 1 s 1))))
                :rthm-seq-map '((1 ((flt (seq1 seq1 seq2 seq1 seq1 seq2)))))))))
    (print (has-mark (get-note mini 1 1 'flt) 'fff))
    (print (swap-marks mini nil nil nil 'pp 'fff))
    (has-mark (get-note mini 1 1 'flt) 'fff))

NIL
4
=> (FFF)
```

SYNOPSIS:

```
(defmethod swap-marks ((sc slippery-chicken) start-bar end-bar
                       players old-marks new-marks)
```

19.87   slippery-chicken-edit/thin

[slippery-chicken-edit] [Methods]

DATE:

 19th September 2018, Heidhausen

DESCRIPTION:

 Thin out events in a slippery-chicken object using an activity curve. This
 turns existing notes into rests. As this is an expensive method but may be
 called more than once, it is up to the user to call consolidate-rests and/or
 update-slots when ready.

ARGUMENTS:

- the (fully-initialised) slippery-chicken object

OPTIONAL ARGUMENTS:

keyword arguments
- :start-bar. The bar to start at. Default = NIL = 1
- :end-bar. The bar to end at. Default = NIL = last bar in piece/object
- :players. A symbol or list of symbols for the players to be processed.
- :curve. A list of breakpoints representing the level of activity. Y-values
  should range over 0 (all events forced to rests) to 10 (no rests
  created). X-values may range over any arbitrary scale but NB that this
  will be rescaled to map over the number of bars in the whole piece,
  despite the start-bar/end-bar arguments. However this rescaling can be
  avoided via the next argument. Default = '(0 1 100 10)
- :rescale-curve. Whether to process the x-values of :curve to range over
  the number of bars in the piece. Default = T = rescale.

RETURN VALUE:

- the processed slippery-chicken object

SYNOPSIS:

```
(defmethod thin ((sc slippery-chicken) &key start-bar end-bar players
                                        (curve '(0 1 100 10))
                                        (rescale-curve t))
```

19.88   slippery-chicken-edit/tie

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Add a tie to a specified event object. The new tie will be placed starting
from the specified event object and spanning to the next event object. If
the next event object does not have the same pitch, its pitch will be
changed to that of the first event object.

An optional argument allows the user to adjust the steepness of the tie's
curvature.

NB: This method will not automatically update ties in MIDI output. To make
    sure that MIDI ties are also updated, use the handle-ties method.

NB: If the next event object is a rest and not a note, an error will be
    produced.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the tie is to be
  placed.
- An integer that is the number of the note to which the tie is to be
  attached.
- The ID of the player whose part is to be changed.

OPTIONAL ARGUMENTS:

- A positive or negative decimal number to indicate the steepness of the
  tie's curvature.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4))))
        :set-map '((1 (1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q s s (s) s))
                                :pitch-seq-palette ((1 1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1)))))))))
  (tie mini 2 1 'vn)
  (tie mini 3 2 'vn)
  (tie mini 4 2 'vn -.5))

=> T
```

SYNOPSIS:

```
(defmethod tie ((sc slippery-chicken) bar-num note-num player
                &optional curvature)
```

19.89   slippery-chicken-edit/tie-all-last-notes-over-rests

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Extend the duration of the last note of any bar that precedes a bar which
starts with a rest in the specified region, such that the rest that begins
the next measure is changed to a note and the last note of the first
measure is tied to it.

NB: This method will not automatically update ties in MIDI output. To make
    sure that MIDI ties are also updated, use the handle-ties method.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the first bar in which changes are to be made.
 - An integer that is the last bar in which changes are to be made.
 - A player ID or list of player IDs.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :to-next-attack. T or NIL to indicate whether ties are to extend over
   only full bars of rest or also over partial bars (until the next attacked
   note). T = until the next attacked note. Default = T.
 - :tie-next-attack. T or NIL to indicate whether the new tied notes created
   should also be further extended over the next attacked note if that note
   has the same pitch as the starting note of the tie. T = also tie next
   attacked note if same pitch. Default = NIL.
 - :auto-beam. T or NIL to indicate whether the new events should be
   automatically beamed after placement. T = automatically beam.
   Default = NIL.
 - :last-rhythm. NIL or a rhythmic duration. If a rhythmic duration, the
   last duration of the tie will be forced to this length. Useful, for
   example, when tying into a rest bar without filling that whole
   bar. NIL = fill the bar with a tied note. Default = NIL.
 - :update. T or NIL to indicate whether all slots for all events in the
   piece should be updated. This is an expensive operation so set to NIL if
   you plan on calling other similar methds, and call (update-slots sc)
   explicitly at the end. Default = T.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
```

```
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (viola :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 f4 g4 a4 c5 d5 f5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e (e) e e (e) (e) e e)
                                 ((w))
                                 ((h.) q)
                                 ((w))
                                 ((w))
                                 ((e) e h.))
                                :pitch-seq-palette ((1 2 3 4 5 6 7 7)))))
        :rthm-seq-map '((1 ((vn (1 1 1))
                            (va (1 1 1))
                            (vc (1 1 1)))))))))
  (tie-all-last-notes-over-rests mini 2 6 'vn)
  (tie-all-last-notes-over-rests mini 9 12 'vn :auto-beam t)
  (tie-all-last-notes-over-rests mini 3 5 '(va vc) :to-next-attack nil)
  (tie-all-last-notes-over-rests mini 9 12 'vc :tie-next-attack t)
  (tie-all-last-notes-over-rests mini 13 15 'vn :last-rhythm 'e))


=> NIL
```

SYNOPSIS:

```
(defmethod tie-all-last-notes-over-rests
    ((sc slippery-chicken)
     start-bar end-bar players
     &key
       ;; use up all rests until next attack or (if nil)
       ;; just the rest bars?
       (to-next-attack t)
       ;; if the next attack is the same note/chord as
       ;; the previous, tie to it too?
       (tie-next-attack nil)
       (last-rhythm nil)
       (update t)
       (auto-beam nil))
```

19.90   slippery-chicken-edit/tie-over-all-rests

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Extend the durations of all notes that immediately precede rests in the
specified region by changing the rests to notes and tying the previous notes
to them.

NB: This method will not automatically update ties in MIDI output. To make
    sure that MIDI ties are also updated, use the handle-ties method.

ARGUMENTS:

- A slippery-chicken object.
- The ID of the player whose part is to be changed, or a list of players or
  NIL to indicate that we should process all players
- An integer that is the number of the first bar in which notes are to be
  tied over rests.
- An integer that is the number of the last bar in which notes are to be
  tied over rests. NB: This argument does not necessarily indicate the bar
  in which the ties will stop, but rather the last bar in which a tie will
  be begun; the ties created may extend into the next bar.

OPTIONAL ARGUMENTS:

keyword arguments:
- :start-note. An integer that is the number of the first attacked note
  (not counting rests) in the given start-bar for which ties can be placed.
- :end-note. An integer that is the number of the last attacked note (not
  counting rests) in the given end-bar for which ties can be placed.
  NB: This argument does not necessarily indicate the note on which the
  ties will stop, but rather the last not on which a tie can begin; the
  ties created may extend to the next note.
- :auto-beam. T or NIL to indicate whether the method should automatically
  place beams for the notes of the affected measure after the ties over
  rests have been created. T = automatically beam. Default = NIL.
- :consolidate-notes. T or NIL to indicate whether the tied note are to be
  consolidated into single rhythmic units of longer durations after the
  ties over rests have been created. T = consolidate notes. Default = NIL.
- :update. T or NIL to indicate whether all slots for all events in the
  piece should be updated. This is an expensive operation so set to NIL if
  you plan on calling other similar methds, and call (update-slots sc)
  explicitly at the end. Default = T.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4))))
        :set-map '((1 (1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) (q) e (s) s))
                                :pitch-seq-palette ((1 2)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1 1)))))))))
  (tie-over-all-rests mini 'vn 2 3 :start-note 2 :auto-beam t)
  (tie-over-all-rests mini 'vn 5 6 :end-note 1 :consolidate-notes t))

  => T
```

SYNOPSIS:

```
(defmethod tie-over-all-rests ((sc slippery-chicken) players
                               start-bar end-bar
                               &key
                                  (start-note 1)
                                  (end-note 9999999)
                                  (auto-beam nil)
                                  (update t)
                                  (consolidate-notes nil))
```

19.91   slippery-chicken-edit/tie-over-rest-bars

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Extend the duration of the last note in a specified bar by changing
immediately subsequent full-rest bars to notes of the same pitch and tying
them to that note.

NB: This method will not automatically update ties in MIDI output. To make
    sure that MIDI ties are also updated, use the handle-ties method.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the last note is to be
  tied.
- An ID or list of IDs of the players whose parts are to be modified.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :end-bar. An integer or NIL. If an integer, this is the number of the
  last bar of full-rests that is to be changed to a note. This can be
  helpful for tying into passages of multiple bars of full-rest.
- :tie-next-attack. T or NIL to indicate whether the new tied notes created
  should also be further extended over the next attacked note if that note
  has the same pitch as the starting note of the tie. T = also tie next
  attacked note if same pitch. Default = NIL.
- :to-next-attack. T or NIL to indicate whether ties are to extend over
  only full bars of rest or also over partial bars (until the next attacked
  note). T = until the next attacked note. Default = T.
- :auto-beam. T or NIL to indicate whether the method should automatically
  place beams for the notes of the affected measure after the ties over
  rests have been created. T = automatically beam. Default = NIL.
- :last-rhythm. NIL or a rhythmic duration. If a rhythmic duration, the
  last duration of the tie will be forced to this length. Useful, for
  example, when tying into a rest bar without filling that whole
  bar. NIL = fill the bar with a tied note. Default = NIL.
- :update. T or NIL to indicate whether all slots for all events in the
  piece should be updated. This is an expensive operation so set to NIL if
  you plan on calling other similar methds, and call (update-slots sc)
  explicitly at the end. Default = T.
```

RETURN VALUE:

 Returns t.

EXAMPLE:

```
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vn (violin :midi-channel 1))
                      (va (viola :midi-channel 2))
                      (vc (cello :midi-channel 3))))
         :set-palette '((1 ((c4 d4 e4))))
         :set-map '((1 (1 1)))
         :rthm-seq-palette '((1 ((((2 4) (q) e (s) s)
                                  ((h))
                                  ((s) e. e e)
                                  ((h))
                                  ((h))
                                  ((e) q s (s)))
                                 :pitch-seq-palette ((1 2 2 1 3 3 1)))))
         :rthm-seq-map '((1 ((vn (1 1))
                             (va (1 1))
```

```
                                (vc (1 1))))))))))
  (tie-over-rest-bars mini 1 'vn :end-bar 2)
  (tie-over-rest-bars mini 3 'va :end-bar 5)
  (tie-over-rest-bars mini 3 '(vn vc) :end-bar 6 :tie-next-attack t)
  (tie-over-rest-bars mini 7 'vc
                         :end-bar 9
                         :to-next-attack t
                         :auto-beam t)
  (tie-over-rest-bars mini 9 'vn :end-bar 11 :last-rhythm 'e))

  => t
```

SYNOPSIS:

```
(defmethod tie-over-rest-bars ((sc slippery-chicken) bar-num players
                                &key (end-bar nil) ;; num of empty bars
                                   (tie-next-attack nil)
                                   (to-next-attack t)
                                   (last-rhythm nil)
                                   (update t)
                                   (auto-beam nil))
```

19.92   slippery-chicken-edit/tie-over-rests

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Extend the duration of a specified note that precedes a rest by changing
the rest to a note with the same pitch and adding a tie between them.

NB: This method will not automatically update ties in MIDI output. To make
    sure that MIDI ties are also updated, use the handle-ties method.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar in which the note is located.
 - An integer that is the number of the note within that bar which is to be
   extended. This number is 1-based and also counts already tied notes. If
   NIL, then the last note in the bar will be used.
 - The ID of the player whose part is to be modified.

OPTIONAL ARGUMENTS:

```
keyword arguments
- :end-bar. An integer that is the number of the last bar into which the
  tie is to extend. This can be helpful if the user wants to tie into only
  the first of several consecutive full-rest bars.
- :auto-beam. T or NIL to indicate whether the method should automatically
  beam the beats of the modified bars after the ties have been added.
  T = automatically beam. Default = NIL.
- :consolidate-notes. T or NIL to indicate whether the method should
  consolidate tied notes into single rhythm units of longer duration.
  T = consolidate. Default = T.
- :update. T or NIL to indicate whether all slots for all events in the
  piece should be updated. This is an expensive operation so set to NIL if
  you plan on calling other similar methds, and call (update-slots sc)
  explicitly at the end. Default = T.
```

RETURN VALUE:

```
Returns T.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) (q) e (s) s)
                                 ((h))
                                 ((s) e. (e) e)
                                 ((h))
                                 ((h))
                                 ((e) q s (s)))
                                :pitch-seq-palette ((1 2 2 3 3 1)))))
        :rthm-seq-map '((1 ((vn (1 1 1)))))))))
  (tie-over-rests mini 1 2 'vn)
  (tie-over-rests mini 7 1 'vn)
  (tie-over-rests mini 9 2 'vn :end-bar 10)
  (tie-over-rests mini 13 1 'vn :auto-beam t :consolidate-notes nil))

  => T
```

SYNOPSIS:

```
(defmethod tie-over-rests ((sc slippery-chicken) bar-num note-num player
                           &key end-bar auto-beam (consolidate-notes t)
                                (update t))
```

19.93   slippery-chicken-edit/tie-repeated-notes

[slippery-chicken-edit] [Methods]

DATE:

 Wed 18 Sep 2019 18:56:39 BST

DESCRIPTION:

 Tie adjacent notes if they are the same pitch or chord.
 NB: this method works on pitches and chords

ARGUMENTS:

 - a slippery chicken object
 - the first bar to start looking
 - the last bar to look in
 - the player or players to tie

OPTIONAL ARGUMENTS:

 - :consolidate - call consolidate-all-notes at the end? Default = t

RETURN VALUE:

 A list of the total number of ties between notes per instrument

EXAMPLE:

```
(let* ((mini (make-slippery-chicken
               '+mini+
               :ensemble '(((pno (piano :midi-channel 1))))
               :staff-groupings '(1)
               :tempo-map '((1 (q 60)))
               :set-palette '((set1 ((fs2 b2 d4 a4 d5 e5 a5 d6)))
                              (set2 ((b2 fs3 d4 e4 a4 d5 e5 a5 d6))))
               :set-map '((1 (set1 set1 set2 set1 set1 set2)))
               :rthm-seq-palette
               '((seq1 ((((4 4) q (q) q q))
                       :pitch-seq-palette (1 1 1)))
                 (seq2 ((((4 4) (e) e q e (e) e e))
                       :pitch-seq-palette (1 1 1 (1) (1)))))
               :rthm-seq-map '((1 ((pno (seq1 seq1 seq2 seq1 seq1 seq2)))))))))
      (tie-repeated-notes mini nil nil nil))

=> (12)
```

SYNOPSIS:

```
(defmethod tie-repeated-notes ((sc slippery-chicken) start-bar end-bar players
                               &key (consolidate t) (check-ties? t))
```

19.94   slippery-chicken-edit/trill

[slippery-chicken-edit] [Methods]

DESCRIPTION:

Attach a trill mark to a specified event object by adding 'BEG-TRILL-A to
the MARKS-BEFORE slot and TRILL-NOTE with the pitch to the MARKS slot. This
method requires a specified trill pitch.

By default trills are set to span from the specified note to the next note,
though the length of the span can be specified using the optional
arguments.

NB: This is a LilyPond-only method and will not affect CMN output.

ARGUMENTS:

 - A slippery-chicken object.
 - The player to whose part the trill is to be added.
 - An integer that is the number of the bar in which the trill is to start.
 - An integer that is the number of the event object in that bar on which
   the trill is to be placed.
 - A note-name symbol that is the pitch of the trill note.

OPTIONAL ARGUMENTS:

 - An integer that is the number of the event object on which the trill span
   is to stop.
 - An integer that is the number of the bar in which the trill span is to
   stop.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
```

```
      (make-slippery-chicken
       '+mini+
       :ensemble '(((vn (violin :midi-channel 1))))
       :set-palette '((1 ((c4 d4 e4))))
       :set-map '((1 (1 1 1 1 1)))
       :rthm-seq-palette '((1 ((((2 4) q. s s))
                                 :pitch-seq-palette ((1 3 2)))))
       :rthm-seq-map '((1 ((vn (1 1 1 1 1))))))))))
  (trill mini 'vn 2 1 'e4)
  (trill mini 'vn 3 1 'e4 3)
  (trill mini 'vn 4 1 'e4 3 5))

=> T
```

SYNOPSIS:

```
(defmethod trill ((sc slippery-chicken) player start-bar start-event trill-note
                  &optional end-event end-bar)
```

19.95   slippery-chicken-edit/unset-cautionary-accidental

[slippery-chicken-edit] [Methods]

DESCRIPTION:

A post-generation editing method: Remove the parentheses from a cautionary
accidental (leaving the accidental itself) by setting the
ACCIDENTAL-IN-PARENTHESES slot of the contained pitch object to NIL.

NB: Since respell-notes is called by default within cmn-display and
    write-lp-data-for-all, that option must be explicitly set to NIL for
    this method to be effective.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar in which the cautionary
  accidental is to be unset.
- An integer that is the number of the note in that bar for which the
  cautionary accidental is to be unset.
- The ID of the player whose part is to be changed.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to unset the cautionary accidental for the

```
   written part only (for transposing instruments).
   T = written only. Default = NIL.
```

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vn (violin :midi-channel 2))))
        :set-palette '((1 ((cs4 ds4 fs4))))
        :set-map '((1 (1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 3 2 1 2 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1))
                            (vn (1 1)))))))))
  (respell-notes mini)
  (unset-cautionary-accidental mini 2 5 'vn)
  (unset-cautionary-accidental mini 2 7 'cl t)
  (cmn-display mini :respell-notes nil))
```

SYNOPSIS:

```
(defmethod unset-cautionary-accidental ((sc slippery-chicken) bar-num note-num
                                        player &optional written)
```

20   sc/spectra

[Modules]

NAME:

 spectra

 File: spectra.lsp

 Class Hierarchy: none (no classes defined)

 Version: 1.0.11

```
Project:           slippery chicken (algorithmic composition)

Purpose:           Definition of the spectral data for piano and other
                   instruments' notes. We'll use this data for
                   the calculation of chord dissonance values in the chord
                   class's calculate-dissonance method.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     July 27th 2015

$$ Last modified: 12:27:57 Tue Feb  2 2016 GMT

SVN ID: $Id: spectra.lsp 5359 2015-07-24 20:53:22Z medward2 $
```

20.1   spectra/get-spectra-al

[spectra] [Functions]

DATE:

August 6th 2015, Edinburgh

DESCRIPTION:

Create an assoc-list of spectral data by doing spectral analysis of all the
sound files contained in a given directory. The aim is to extract the
spectral data for harmonic partials, so we specifically look for ascending
partials as close as possible to integer multiples of the sample sound
file's fundamental frequency, as detected from the file name (see below).

By default we average spectra over three separate readings at 300, 400, and
500 milliseconds. We do this in order to avoid spectral anomalies at any
fixed point in any of the sound files, and of course to skip the onset
transient and any inharmonicities therein.

As the extensive analysis and processing here will take some computation
time, in order to create our default spectral data for calculating a
chord's dissonance or spectral centroid, I've performed the analysis on
some of my local sample libraries. By then using the assoc-list method
print-for-init I've copied the data into separate text files for reading
into the +slippery-chicken-spectra+ global assoc-list and thus making these
available to all slippery-chicken users. By modifying the examples at the
bottom of this file, similar sample directories' spectral data could be
added to this global.

ARGUMENTS:

- the path to the samples directory. This should be something suitable for
passing to the (directory) function so we'll need the extension of the sound
files (see "*.wav" in the example below).
- a function for processing the filenames (as a string, with no directory
or extension e.g. "Stein-R(A0)-V(085)-dB(3478)-T(ML)-M(28)-P(2126-03-01)")
in order to return the MIDI note number of the sample in that file. This
whole approach is predicated on the assumption that sample file names will
contain such pitch data. If that is not the case then get-spectra-al will
be of no use. See the akoustik-piano-name and violin-ensemble-name
functions below for examples.

OPTIONAL ARGUMENTS:

 Keyword arguments:
- :force-harmonics: whether to look for harmonic partials or not. If this
   is nil then the returned spectra will be the strongest by amplitude
   irrespective of their harmonicity. Default = t
- :harmonic-tolerance: the maximum deviation from the exact integer
   partial. Default = 0.15
- :fftsize: The FFT window size for spectral analysis. Default = 4096
- :num-partials: the number of partials to identify and return.
   Default = 12
- :id: The default ID for the resultant assoc-list. Default = 'spectra
- :analysis-points: The times in milliseconds at which to do the spectral
   analysis and extract averages. If no averaging is necessary simply pass
   one starting value as a list. Default = '(300 400 500)

RETURN VALUE:

 An assoc-list object whose keys are the MIDI note numbers associated with
 the spectral information of sample sound files. Each of these is in the
 form of a two element list: a list of fundamental scalers and a list of
 their amplitudes (normalised to 1.0).

EXAMPLE:

```
(get-spectra-al "/path/to/samples/*.wav"
                #'violin-ensemble-name :id 'violin-ensemble)
-->
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 21, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
```

```
NAMED-OBJECT: id: VIOLIN-ENSEMBLE, tag: NIL,
data: (
NAMED-OBJECT: id: 56, tag: NIL,
data: ((0.9889743194880222d0 2.0263850081954717d0 2.9990610385449314d0
        4.024036707283989d0 5.032546599880615d0 6.013292183763263d0
        7.0551329128964655d0 8.049590274637817d0 9.032029362954297d0
        10.017427921485291d0 11.0461040956976d0 12.028523416637086d0)
       (0.1748299156810844d0 0.6666103041159978d0 0.740572572283973d0
        0.37090168062541756d0 0.2529382535743205d0 0.19313279338531672d0
        0.290088638695093d0 0.20008736984066355d0 0.12280861470679348d0
        0.15534581915388804d0 0.11772253740784348d0 0.11703054477270619d0))
...
```

SYNOPSIS:

```
#+clm
(defun get-spectra-al (sample-dir name-fun
                       &key (force-harmonics t) (harmonic-tolerance 0.15)
                         (fftsize 4096) (num-partials 12) (id 'spectra)
                         ;; millisec times of fft analysis
                         (analysis-points '(300 400 500)))
```

21   sc/utilities

[Modules]

NAME:

```
 utilities

 File:              utilities.lsp

 Class Hierarchy:   none: no classes defined

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           Various helper functions of a general nature.

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     June 24th 2002

 $$ Last modified:  15:40:26 Thu Sep 24 2020 CEST
```

SVN ID: Id

21.1   utilities/a-weighting

[utilities] [Functions]

DESCRIPTION:

 Implementation of A-weighting loudness compensation. Formula taken from
 http://en.wikipedia.org/wiki/A-weighting. This doesn't take 1000Hz
 loudness into account, rather it implements the 40-phon Fletcher-Munson
 curve only.

ARGUMENTS:

 The frequency in Hertz for which to find the loudness weighting.

OPTIONAL ARGUMENTS:

 keyword aguments:
 - :expt. A power (exponent) to raise the result to in order to
   tame/exaggerate the curve (make the db weightings less/more
   extreme). This only really makes sense if :linear t though will work
   with db values also of course. Values < 1 result in linear values
   closer to 1 (less extreme). Values > 1 are further from 1. Default = NIL
   i.e. no exponential function.
 - :linear. If T return amplitude values as linear scalers rather than
   logarithmic decibel values. NB If this is NIL then returned values are
   likely to be negative (db) values. Default = T.
 - :invert. As the weighting routine tries to tell us what relative
   loudness we'll perceive given constant amplitudes, low and high
   frequencies will return negative values as we perceive them Xdb less
   than our most sensitive frequency area. If :invert t, just flip this
   negatives to positives so that if :linear T you get a scaler to make
   lower/higher frequences equally loud as the most sensitive frequencies.

RETURN VALUE:

 The linear or db weighting value for the given frequency.

EXAMPLE:

```
;;; Decibels:
(a-weighting 50 :invert nil :linear nil) => -30.274979
```

```
(a-weighting 50 :invert t :linear nil) => 30.274979
;;; Linear amplitude scalers:
(a-weighting 50) => 32.639904
(a-weighting 50 :invert nil) => 0.030637344
;;; Exaggeration:
(a-weighting 50 :expt 1.1) => 46.251286
;;; Smoothing:
(a-weighting 50 :expt .5) => 5.7131343


;;; Looping through the MIDI note range by tritones returning decibel values:
(loop for midi from 0 to 127 by 6
     for freq = (midi-to-freq midi)
     collect (list (midi-to-note midi)
                   (a-weighting freq :linear nil :invert nil)))
=>
((C-1 -76.85258) (FS-1 -65.94491) (C0 -55.819363) (FS0 -46.71565)
 (C1 -38.714867) (FS1 -31.724197) (C2 -25.598646) (FS2 -20.247103)
 (C3 -15.622625) (FS3 -11.657975) (C4 -8.258142) (FS4 -5.358156)
 (C5 -2.9644737) (FS5 -1.1277018) (C6 0.13445985) (FS6 0.8842882) (C7 1.226917)
 (FS7 1.2351798) (C8 0.89729404) (FS8 0.09495151) (C9 -1.3861179)
 (FS9 -3.7814288))

;;; Similar but returning linear amplitude scalers:
(loop for midi from 0 to 127 by 6
     for freq = (midi-to-freq midi)
     collect (list (midi-to-note midi) (a-weighting freq)))
=>
((C-1 6960.316) (FS-1 1982.6475) (C0 617.9711) (FS0 216.6619) (C1 86.246864)
 (FS1 38.56647) (C2 19.051636) (FS2 10.288571) (C3 6.041312) (FS3 3.827355)
 (C4 2.5876594) (FS4 1.8531382) (C5 1.4067719) (FS5 1.1386365) (C6 0.9846389)
 (FS6 0.9032034) (C7 0.8682687) (FS7 0.86744314) (C8 0.9018521) (FS8 0.9891278)
 (C9 1.1730213) (FS9 1.5455086))
```

SYNOPSIS:

```
(defun a-weighting (f &key expt (linear t) (invert t))
```

21.2 utilities/all-members

[utilities] [Functions]

DESCRIPTION:

Find out whether the members of the list given as the second argument are
all present in the list given as the first argument.

ARGUMENTS:

- A list in which the members of the second argument will be sought.
- A list whose members will be sought in the first argument.

OPTIONAL ARGUMENT
- A comparison function.

RETURN VALUE:

 T or NIL.

EXAMPLE:

(all-members '(1 2 3 4 5 6 7) '(1 2 3 7))

=> T

SYNOPSIS:

(defun all-members (list test-list &optional (test #'equal))

21.3   utilities/almost-flatten

[utilities] [Functions]

DATE:

 September 4th 2013

DESCRIPTION:

 Similar to flatten but allows one level of nesting

ARGUMENTS:

 A list with an arbitrary level of nesting.

RETURN VALUE:

 A list with a maximum of one level of nesting

EXAMPLE:

```
(almost-flatten '((1 (2 3 4) (5 (6 7) (8 9 10 (11) 12)) 13) 14 15 (16 17)))
=> (1 (2 3 4) 5 (6 7) 8 9 10 (11) (12) (13) 14 15 (16 17))
```

SYNOPSIS:

```
(defun almost-flatten (nested-list)
```

21.4   utilities/almost-zero

[utilities] [Functions]

DESCRIPTION:

 Return T if a given decimal is within 0.000001 of 0.0.

ARGUMENTS:

 - A number.

OPTIONAL ARGUMENTS:

 - A number that is a user-specified difference for the comparison test.

RETURN VALUE:

 T if the number is within the tolerance difference to zero, otherwise NIL.

EXAMPLE:

```
(almost-zero 0.0000007)
```

```
=> T
```

SYNOPSIS:

```
(defun almost-zero (num &optional (tolerance 0.000001))
```

21.5   utilities/amp2db

[utilities] [Methods]

DESCRIPTION:

 Convert a standard digital amplitude value (>0.0 to 1.0) to a corresponding
 decibel value.

ARGUMENTS:

 - A decimal number between >0.0 and 1.0.

RETURN VALUE:

 A decimal number that is a value in decibel.

EXAMPLE:

(amp2db 0.3)

=> -10.457575

SYNOPSIS:

(defmacro amp2db (amp)

21.6   utilities/amplitude-to-dynamic

[utilities] [Functions]

DESCRIPTION:

 Convert a specified digital amplitude between 0.0 and 1.0 to a
 corresponding dynamic between niente and ffff.

ARGUMENTS:

 - A decimal number between 0.0 and 1.0.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print a warning if the specified
   amplitude is <0.0 or >1.0. T = warn. Default = T.

RETURN VALUE:

 A symbol that is a dynamic level.

EXAMPLE:

(amplitude-to-dynamic 0.3)

=> PP

SYNOPSIS:

(defun amplitude-to-dynamic (amp &optional (warn t))

21.7   utilities/auto-scale-env

[utilities] [Functions]

DATE:

 August 29th 2013

DESCRIPTION:

 Automatically scale both the x and y values of an envelope to fit within
 the given ranges. Normally we'll assume that the minimum and maximum Y
 values are present in the original envelope and so the automatically scaled
 envelope will represent these with the new minimum and maximum
 values. However sometimes an envelope doesn't range over the possible
 extremes, for example (0 .3 100 .6) where the y range is from 0 to 1. If
 this is the case and you need a scaled envelope to take this into account,
 then how is the original envelopes minimum and maximum values to the
 keyword argument :orig-y-range.

ARGUMENTS:

 - The envelope: a list of x y pairs

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :x-min: The new minimum (starting) x value
 - :x-max: The new maximum (last) x value
 - :y-min: The new minimum (not necessarily starting!) y value
 - :y-max: The new maximum (not necessarily starting!) y value
 - :orig-y-range: a two-element list specifying the original envelope's
   minimum and maximum values (see above).

RETURN VALUE:

 The new envelope (list).

EXAMPLE:

```
(auto-scale-env '(0 0 10 1))
=>
(0.0 0.0 100.0 10.0)

(auto-scale-env '(-1 0 .3 -3 1 1) :y-min 5 :y-max 6 :x-min 2)
```

```
=>
(2.0 5.75 65.7 5.0 100.0 6.0))

(auto-scale-env '(0 1 5 1.5 7 0 10 1) :y-min -15 :y-max -4)
=>
(0.0 -7.6666665 50.0 -4.0 70.0 -15.0 100.0 -7.6666665))

(auto-scale-env '(0 .5 100 .5) :y-min 1 :y-max 2)
=> (0.0 1.0 100.0 1.0)

(auto-scale-env '(0 .5 100 .5) :y-min 1 :y-max 2 :orig-y-range '(0 1))
=> (0.0 1.5 100.0 1.5)
```

SYNOPSIS:

```
(defun auto-scale-env (env &key
                             (x-min 0.0) (x-max 100.0)
                             (y-min 0.0) (y-max 10.0)
                             orig-y-range)
```

21.8   utilities/between

[utilities] [Functions]

DESCRIPTION:

Return a random number between two specified numbers. If the two numbers are
integers, the random selection is inclusive. If either are floating-point
(decimal) numbers, the result will be a float between the first (inclusive)
and just less than the second (i.e. exclusive).

ARGUMENTS:

- A first, lower, number.
- A second, higher, number.

NB: The first number must always be lower than the second.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the random seed should be fixed.
- T or NIL to indicate whether, when fixed-random is set to T, we should
  reset the random number generator (to guarantee the same random
  sequences). This would generally only be called once, perhaps at the
  start of a generation procedure.

RETURN VALUE:

 An integer if both numbers are integers, or a float if one or both are
 decimal numbers.

EXAMPLE:

```
;;; Using the defaults. This will produce a different result each time.
(loop repeat 10 collect (between 1 100))

=> (43 63 26 47 28 2 99 93 66 23)

;;; Setting fixed-random to T and using zerop to reset the random when i is 0
(loop repeat 5
   collect (loop for i from 0 to 9 collect (between 1 100 t (zerop i))))

=> ((93 2 38 81 43 19 70 18 44 26) (93 2 38 81 43 19 70 18 44 26)
    (93 2 38 81 43 19 70 18 44 26) (93 2 38 81 43 19 70 18 44 26)
    (93 2 38 81 43 19 70 18 44 26))
```

SYNOPSIS:

```
(defun between (low high &optional fixed-random restart)
```

21.9   utilities/between-extremes

[utilities] [Functions]

DATE:

 April 23rd 2016, Edinburgh

DESCRIPTION:

 Given a <progress> value of between 0 and 1, we'll return whatever that
 proportion is of the difference between <min> and <max> added to min. NB No
 randomness here.

ARGUMENTS:

```
 - the minimum value (returned when <progress> is 0.0)
 - the maximum value (returned when <progress> is 1.0)
 - a value between
```

RETURN VALUE:

 a number between min and max

EXAMPLE:

```
(BETWEEN-EXTREMES 0.5 1 0.5)
==> 0.75
(BETWEEN-EXTREMES 0.5 1 0.9)
==> 0.95
```

SYNOPSIS:

```
(defun between-extremes (min max progress)
```

21.10   utilities/combine-into-symbol

[utilities] [Functions]

DESCRIPTION:

 Combine a sequence of elements of any combination of type string, number,
 or symbol into a symbol.

ARGUMENTS:

 - A sequence of elements.

RETURN VALUE:

 A symbol as the primary value, with the length of that symbol as a
 secondary value.

EXAMPLE:

```
(combine-into-symbol "test" 1 'a)
```

```
=> TEST1A, 6
```

SYNOPSIS:

```
(defun combine-into-symbol (&rest params)
```

21.11   utilities/db2amp

[utilities] [Functions]

DESCRIPTION:

Convert a decibel value to a standard digital amplitude value (>0.0 to 1.0), whereby 0dB = 1.0.

ARGUMENTS:

- A number that is a value in decibel.

RETURN VALUE:

A decimal number between >0.0 and 1.0.

EXAMPLE:

```
(db2amp -3)
```

```
=> 0.70794576
```

SYNOPSIS:

```
(defmacro db2amp (db)
```

21.12   utilities/decimal-places

[utilities] [Functions]

DATE:

19-Mar-2012

DESCRIPTION:

Round the given number to the specified number of decimal places.

ARGUMENTS:

- A number.
- An integer that is the number of decimal places to which to round the given number.

RETURN VALUE:

A decimal number.

EXAMPLE:

```
(decimal-places 1.1478349092347 2)
```

```
=> 1.15
```

SYNOPSIS:

```
(defun decimal-places (num places)
```

21.13   utilities/decimate-env

[utilities] [Functions]

DESCRIPTION:

Reduce the number of x,y pairs in an envelope. In every case the envelope
is first stretched along the x-axis to fit the new number of points
required. Then we proceed by one of three methods:

1) average: for every new output x value, interpolate 100 times from -0.5
to +0.5 around the point, then average the y value. This will catch
clustering but round out spikes caused by them

2) points: also an averaging method but only using the existing points in
the original envelope (unless none is present for a new x value, whereupon
interpolation is used): Take an average of the (several) points nearest the
new output point. This might not recreate the extremes of the original
envelope but clustering is captured, albeit averaged.

3) interpolate: for each new output point, interpolate the new y value from
the original envelope. This will leave out details in the case of
clustering, but accurately catch peaks if there are enough output points.

In each case we create an even spread of x values, rather than clustering
where clusters exist in the original.

ARGUMENTS:

- the original envelope (list of x,y values on any scales).
- the number of points required in the output list.

OPTIONAL ARGUMENTS:

- the method to be applied (symbol): 'points, 'average, 'interpolate.
  Default = 'points.

RETURN VALUE:

 A list representing the x,y values of the new envelope

EXAMPLE:

```
(decimate-env '(0 0 4 4 5 5 5.1 5.1 5.3 1 5.6 5.6 6 6 10 10) 6)
=>
(0.0 0.0 1 2.0 2 4.5 3 4.425 4 8.0 5.0 10.0)
```

SYNOPSIS:

```
(defun decimate-env (env num-points &optional (method 'points))
```

21.14   utilities/down-up

[utilities] [Functions]

DATE:

 May 8th 2016

DESCRIPTION:

 This is a routine used in morphing maps but may be useful elsewhere. It
 interpolates between two numbers over a given number of steps before
 returning back to the first number.

ARGUMENTS:

 - the number of steps over which the procedure should interpolate

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :down. T or NIL: whether to first descend before ascending. Default = T
 - :up. T or NIL: whether to ascend after descending. Default = T
 - :start. The number to start at. Default = 1.0
 - :target. The number to interpolate towards. Default = 0.0.
 - :cons. Whether to return :start as the first number in the result list.
 - :butlast. Whether to omit the :start when ascending. Default = T.

RETURN VALUE:

A list of numbers.

EXAMPLE:

```
(mapcar #'(lambda (x) (decimal-places x 2)) (down-up 20))
--> (0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0 0.09 0.18 0.27 0.36 0.45 0.55 0.64
     0.73 0.82 0.91)
(mapcar #'(lambda (x) (decimal-places x 2)) (down-up 20 :cons t))
--> (1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8
     0.9)
(mapcar #'(lambda (x) (decimal-places x 2)) (down-up 20 :butlast nil))
--> (0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
     1.0)
```

SYNOPSIS:

```
(defun down-up (steps &key (down t) (up t) (start 1.0d0) (target 0.0d0)
                      (cons nil) (butlast t))
```

21.15   utilities/dynamic-to-amplitude

[utilities] [Functions]

DESCRIPTION:

Convert a symbol that is a dynamic level between niente and ffff to a
corresponding digital amplitude value between 0.0 and 1.0.

ARGUMENTS:

 - A symbol that is a dynamic level between niente and fff.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print a warning when the symbol specified
   is not recognized as a dynamic. T = warn. Default = T.

RETURN VALUE:

 A decimal number between 0.0 and 1.0.

EXAMPLE:

```
(dynamic-to-amplitude 'fff)
```

```
=> 0.9
```

SYNOPSIS:

```
(defun dynamic-to-amplitude (dynamic &optional (warn t))
```

21.16   utilities/econs

[utilities] [Functions]

DESCRIPTION:

 Add a specified element to the end of an existing list.

ARGUMENTS:

 - A list.
 - An element to add to the end of the list.

RETURN VALUE:

 A new list.

EXAMPLE:

```
(econs '(1 2 3 4) 5)

=>  '(1 2 3 4 5)
```

SYNOPSIS:

```
(defun econs (list new-back)
```

21.17   utilities/env-plus

[utilities] [Functions]

DESCRIPTION:

 Increase all y values of a given list of break-point pairs by a specified
 amount.

ARGUMENTS:

 - An envelope in the form of a list of break-point pairs.
 - A number that is the amount by which all y values of the given envelope
   are to be increased.

RETURN VALUE:

 A list of break-point pairs.

EXAMPLE:

(env-plus '(0 0 25 11 50 13 75 19 100 23) 7.1)

=> (0 7.1 25 18.1 50 20.1 75 26.1 100 30.1)

SYNOPSIS:

(defun env-plus (env add)

21.18   utilities/env-symmetrical

[utilities] [Functions]

DESCRIPTION:

 Create a new list of break-point pairs that is symmetrical to the original
 around a specified center. If no center is specified, the center value
 defaults to 0.5

ARGUMENTS:

 - An envelope in the form of a list of break-point pairs.

OPTIONAL ARGUMENTS:

 - A number that is the center value around which the values of the
   new list are to be symmetrical.
 - A number that is to be the minimum value for the y values returned.
 - A number that is to be the maximum value for the y values returned.

RETURN VALUE:

 An envelope in the form of a list of break-point pairs.

EXAMPLE:

;;; Default center is 0.5
(env-symmetrical '(0 0 25 11 50 13 75 19 100 23))

```
=> (0 1.0 25 -10.0 50 -12.0 75 -18.0 100 -22.0)

;; Specifying a center of 0
(env-symmetrical '(0 0 25 11 50 13 75 19 100 23) 0)

=> (0 0.0 25 -11.0 50 -13.0 75 -19.0 100 -23.0)

;;; Specifying minimum and maximum y values for the envelope returned
(env-symmetrical '(0 0 25 11 50 13 75 19 100 23) 0 -20 -7)

=> (0 -7 25 -11.0 50 -13.0 75 -19.0 100 -20)
```

SYNOPSIS:

```
(defun env-symmetrical (env &optional (centre .5)
                            (min most-negative-double-float)
                            (max most-positive-double-float))
```

21.19   utilities/env2gnuplot

[utilities] [Functions]

DATE:

 24th December 2013

DESCRIPTION:

```
 Write a data file of x,y envelope values for use with gnuplit.  Once called
 start gnuplot and issue commands such as:
 gnuplot> set terminal postscript default
 gnuplot> set output '/tmp/env.ps'
 gnuplot> plot '/tmp/env.txt' with lines.
```

ARGUMENTS:

 - The envelope as the usual list of x y pairs

OPTIONAL ARGUMENTS:

 - The pathname of the data file to write. Default = "/tmp/env.txt".

RETURN VALUE:

 Always T

SYNOPSIS:

```
(defun env2gnuplot (env &optional (file "/tmp/env.txt"))
```

21.20   utilities/envelope-boundaries

[utilities] [Functions]

DESCRIPTION:

Find sharp changes in envelope values. These are defined as when a y value
rises or falls over 30% (by default) of it's overall range within 5%
(again, by default) of its overall x axis range.

ARGUMENTS:

The envelope (a list of x y pairs).

OPTIONAL ARGUMENTS:

- jump-threshold: the minimum percentage change in y value that is deemed a
  sharp change.
- steepness-min: the maximum percentage of the overall x axis that
  constitutes a 'quick' change.

RETURN VALUE:

A list of x values at which boundaries are deemed to lie.

EXAMPLE:

```
(ENVELOPE-BOUNDARIES '(0 10 20 10 21 3 25 4 26 9 50 7 51 1 55 2 56 7 70 10
                       100 10))
--> (21 26 51 56)
```

SYNOPSIS:

```
(defun envelope-boundaries (envelope &optional (jump-threshold 30)
                              (steepness-min 5))
```

21.21   utilities/equal-within-tolerance

[utilities] [Functions]

DESCRIPTION:

Test whether the difference between two decimal numbers falls within a
specified tolerance.

This test is designed to compensate for calculation discrepancies caused by
floating-point errors (such as 2.0 vs. 1.9999997), in which the equations
should yield equal numbers. It is intended to be used in place of = in such
circumstances.

ARGUMENTS:

 - A first number.
 - A second number.

OPTIONAL ARGUMENTS:

 - A decimal value that is the maximum difference allowed between the two
   numbers that will still return T. Default = 0.000001d0.

RETURN VALUE:

 T if the two tested numbers are equal within the specified tolerance,
 otherwise NIL.

EXAMPLE:

```
;; An example of floating-point error
(loop for i from 0.0 below 1.1 by 0.1 collect i)

=> (0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.70000005 0.8000001 0.9000001 1.0000001)

;; Using =
(loop for i from 0.0 below 1.1 by 0.1
   for j in '(0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0)
   collect (= i j))

=> (T T T T T T T NIL NIL NIL NIL)

;; Using equal-within-tolerance
(loop for i from 0.0 below 1.1 by 0.1
   for j in '(0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0)
   collect (equal-within-tolerance i j))

=> (T T T T T T T T T T T)
```

SYNOPSIS:

```
(defun equal-within-tolerance (a b &optional (tolerance 0.000001d0))
```

21.22 utilities/exaggerate-env

[utilities] [Functions]

DESCRIPTION:

 Makes the y values in an envelope more radically pushed towards its
 extremes. Y values below the mid-point will be pushed downwards; those
 above will be pushed upwards. The opposite can be accomplished by making
 the exponent argument > 1 (see below).

ARGUMENTS:

 - the envelope: a list of numbers representing an envelope: x y pairs
 - the exponent: this determines the amount of
   exaggeration. Counterintuitively perhaps, the lower values are than 1 the
   more exaggeration takes place. Values > 1 will mean the opposite:
   understated y values, if you will.

OPTIONAL ARGUMENTS:

 - easy-expt: because of the counterintuitive nature of the exponent, you
   can pass values between -10 and +10 if this third argument is T. This
   will be scaled to useful though not over-extreme exponents of 1.9 (-10)
   to .1 (+10) with 0 equating to an exponent of 1, i.e. no change.

RETURN VALUE:

 The new exaggerated envelope (a list).

EXAMPLE:

```
(exaggerate-env '(0 0 50 .8 100 1) 1.9)
--> (0 0.0 50 0.6894338 100 1.0)
(exaggerate-env '(0 0 50 .8 100 1) .1)
--> (0 0.0 50 0.9751001 100 1.0)
(exaggerate-env '(0 0 50 .8 100 1) -10)
--> (0 0.0 50 83.19083 100 1.0)
(exaggerate-env '(0 0 50 .8 100 1) -10 t)
--> (0 0.0 50 0.6894338 100 1.0)
(exaggerate-env '(0 0 50 .8 100 1) 0 t)
--> (0 0.0 50 0.8 100 1.0)
(exaggerate-env '(0 0 50 .8 100 1) 10 t)
--> (0 0.0 50 0.9751001 100 1.0)
```

SYNOPSIS:

```
(defun exaggerate-env (env expt &optional easy-expt)
```

21.23   utilities/factor

[utilities] [Functions]

DESCRIPTION:

Boolean test to check if a specified number is a multiple of a second
specified number.

ARGUMENTS:

- A number that will be tested to see if it is a multiple of the second
  number.
- A second number that is the base number for the factor test.

RETURN VALUE:

T if the first number is a multiple of the second number, otherwise NIL.

EXAMPLE:

(factor 14 7)

=> T

SYNOPSIS:

(defun factor (num fac)

21.24   utilities/flatten

[utilities] [Functions]

DESCRIPTION:

Return a list of nested lists of any depth as a flat list.

ARGUMENTS:

- A list of nested lists.

RETURN VALUE:

A flat list.

EXAMPLE:

```
(flatten '((1 (2 3 4) (5 (6 7) (8 9 10 (11) 12)) 13) 14 15 (16 17)))

=> (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
```

SYNOPSIS:

```
(defun flatten (nested-list)
```

21.25   utilities/force-length

[utilities] [Functions]

DATE:

```
 03-FEB-2011
```

DESCRIPTION:

```
 Create a new a list of a specified new length by adding or removing items
 at regular intervals from the original list. If adding items and the list
 contains numbers, linear interpolation will be used, but only between two
 adjacent items; i.e. not with a partial increment.

 NB: The function can only create new lists that have a length between 1 and
     1 less than double the length of the original list.
```

ARGUMENTS:

```
 - A flat list.
 - A number that is the new length of the new list to be derived from the
   original list. This number must be a value between 1 and 1 less than
   double the length of the original list.
```

RETURN VALUE: EXAMPLE:

```
;;; Shortening a list
(force-length (loop for i from 1 to 100 collect i) 17)

=> (1 7 13 20 26 32 39 45 51 57 63 70 76 82 89 95 100)

;;; Lengthening a list
(force-length (loop for i from 1 to 100 collect i) 199)
```

```
=> (1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10 10.5 11 11.5 12
   12.5 13 13.5 14 14.5 15 15.5 16 16.5 17 17.5 18 18.5 19 19.5 20 20.5 21
   21.5 22 22.5 23 23.5 24 24.5 25 25.5 26 26.5 27 27.5 28 28.5 29 29.5 30
   30.5 31 31.5 32 32.5 33 33.5 34 34.5 35 35.5 36 36.5 37 37.5 38 38.5 39
   39.5 40 40.5 41 41.5 42 42.5 43 43.5 44 44.5 45 45.5 46 46.5 47 47.5 48
   48.5 49 49.5 50 50.5 51 51.5 52 52.5 53 53.5 54 54.5 55 55.5 56 56.5 57
   57.5 58 58.5 59 59.5 60 60.5 61 61.5 62 62.5 63 63.5 64 64.5 65 65.5 66
   66.5 67 67.5 68 68.5 69 69.5 70 70.5 71 71.5 72 72.5 73 73.5 74 74.5 75
   75.5 76 76.5 77 77.5 78 78.5 79 79.5 80 80.5 81 81.5 82 82.5 83 83.5 84
   84.5 85 85.5 86 86.5 87 87.5 88 88.5 89 89.5 90 90.5 91 91.5 92 92.5 93
   93.5 94 94.5 95 95.5 96 96.5 97 97.5 98 98.5 99 99.5 100)
```

SYNOPSIS:

```
(defun force-length (list new-len)
```

21.26   utilities/get-clusters

[utilities] [Functions]

DESCRIPTION:

Takes a list with (ascending) numbers and creates sublists of those numbers
within <threshold> of each other.

ARGUMENTS:

A list of (ascending) numbers. NB Though the numbers don't have to be in
ascending order, the design application of the function makes most sense if
they are.

OPTIONAL ARGUMENTS:

The maximum distance between two numbers in order for them to be considered
as part of the same cluster.

RETURN VALUE:

A list with clusters in sublists.

EXAMPLE:

```
(get-clusters '(24 55 58 59 60 81 97 102 106 116 118 119 145 149 151 200 210
```

```
                  211 214 217 226 233 235 236 237 238 239 383 411 415 419))
--> (24 (55 58 59 60) 81 (97 102 106) (116 118 119) (145 149 151) 200
        (210 211 214 217) 226 (233 235 236 237 238 239) 383 (411 415 419))

(get-clusters '(0 .1 .3 .7 1.5 1.55 2 4.3 6.3 6.4) 1)
--> ((0 0.1 0.3 0.7 1.5 1.55 2) 4.3 (6.3 6.4))

(get-clusters '(0 .1 .3 .7 1.5 1.55 2 4.3 6.3 6.4) 0.5)
--> ((0 0.1 0.3 0.7) (1.5 1.55 2) 4.3 (6.3 6.4))
```

SYNOPSIS:

```
(defun get-clusters (list &optional (threshold 5))
```

21.27   utilities/get-harmonics

[utilities] [Functions]

DESCRIPTION:

```
Return a list of the harmonic partial frequencies in Hertz from a
specified (usually fundamental) frequency.
```

ARGUMENTS:

```
- A number that is the fundamental or starting frequency in Hertz.
```

OPTIONAL ARGUMENTS:

```
keyword arguments
- :start-partial. An integer that is the number of the first harmonic
  partial to return. Default = 1.
- :min-freq. A number that is the lowest frequency in Hertz to
  return. Default = 20.
- :max-freq. A number that is the highest frequency in Hertz to
  return. Default = 20000.
- :start-freq-is-partial.  Rather than treating the first argument as the
  fundamental, treat it as the partial number indicated by this argument.
  Default = 1.
- :max-results.  The maximum number of harmonics to return.  Default =
  most-positive-fixnum
- :skip. The increment for the harmonics.  If 1, then we ascend the
  harmonics series one partial at a time; 2 would mean skipping every other
  Default = 1.
- :pitches. Return a list of pitch objects instead of frequencies. Default =
```

```
   NIL.
- :notes. Return a list of 2-element sublists: note symbols in the
  chromatics scale, with cent deviations
```

RETURN VALUE:

```
A list of numbers that are the frequencies in Hertz of harmonic partials
above the same fundamental frequency, or with the respective keyword, as
pitch objects or note symbols
```

EXAMPLE:

```
;;; Get the first 15 harmonic partials above a fundamental pitch of 64 Hertz,
;;; starting with partial 2, and specifying an upper cut-off of 1010 Hz.

(get-harmonics 63 :start-partial 2 :max-freq 1010)

=> (126 189 252 315 378 441 504 567 630 693 756 819 882 945 1008)
```

SYNOPSIS:

```
(defun get-harmonics (start-freq &key (start-partial 1) (min-freq 20)
                                   (start-freq-is-partial 1) (max-freq 20000)
                                   (skip 1)
                                   pitches notes
                                   (max-results most-positive-fixnum))
```

21.28   utilities/get-sublist-indices

[utilities] [Functions]

DESCRIPTION:

```
Get the starting position of sublists within a list as though the complete
set of items were a flat list.
```

ARGUMENTS:

```
- A list of lists.
```

RETURN VALUE:

```
A list of integers that are the indices of the sublists.
```

EXAMPLE:

```
(get-sublist-indices '((1 2) (3 4 5 6) (7 8 9) (10 11 12 13 14) (15)))
```

```
=> (0 2 6 9 14)
```

SYNOPSIS:

```
(defun get-sublist-indices (list)
```

21.29   utilities/get-sublist-lengths

[utilities] [Functions]

DESCRIPTION:

 Get the lengths of all sublists in a given list.

ARGUMENTS:

 - A list of lists.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to first remove zeros caused by empty
   sublists from the result.

RETURN VALUE:

 A list of integers.

EXAMPLE:

```
;; Straightforward usage allows zeros in the result
(get-sublist-lengths '((1 2) (3 4 5 6) (7 8 9) (10 11 12 13 14) ()))
```

```
=> (2 4 3 5 0)
```

```
;; Setting the optional argument to T removes zeros from the result
```

```
(get-sublist-lengths '((1 2) (3 4 5 6) (7 8 9) (10 11 12 13 14) ()) t)
```

```
=> (2 4 3 5)
```

SYNOPSIS:

```
(defun get-sublist-lengths (list &optional (remove-zeros nil))
```

21.30   utilities/hailstone

[utilities] [Functions]

DESCRIPTION:

```
Implementation of the Collatz conjecture (see
http://en.wikipedia.org/wiki/Collatz_conjecture)
```

```
The Collatz conjecture suggests that by starting with a given number, and
if it is even dividing it by two or if it is odd multiplying it by three
and adding one, then repeating with the new result, the process will
eventually always result in one.
```

ARGUMENTS:

```
- A number to start with.
```

RETURN VALUE:

```
A list of the results collected from each iteration starting with the
specified number and ending with one.
```

EXAMPLE:

(hailstone 11)

=> (11 34 17 52 26 13 40 20 10 5 16 8 4 2 1)

SYNOPSIS:

(defun hailstone (n)

21.31   utilities/hz2ms

[utilities] [Functions]

DESCRIPTION:

```
Convert a frequency in Hertz to the equivalent number of milliseconds.
```

ARGUMENTS:

```
- A number that is a Hertz frequency.
```

RETURN VALUE:

 A number that is the millisecond equivalent of the specified Hertz
 frequency.

EXAMPLE:

(hz2ms 261.63)

=> 3.8221915

SYNOPSIS:

(defun hz2ms (hertz)

21.32   utilities/interleave

[utilities] [Functions]

DESCRIPTION:

 Interleave the elements of an aribitrary number of lists. Should the lists
 not be of the same length, this function will only use up as many elements
 as in the shortest list.

ARGUMENTS:

 As many lists as need to be interleaved.

RETURN VALUE:

 A new list of interleaved elements.

EXAMPLE:

(INTERLEAVE '(1 2 3 4 5) '(a b c d) '(x y z))
--> (1 A X 2 B Y 3 C Z)

(INTERLEAVE '(1 2 3 4 5) '(a b c d e) '(v w x y z))
--> (1 A V 2 B W 3 C X 4 D Y 5 E Z)

SYNOPSIS:

(defun interleave (&rest lists)

21.33   utilities/interpolate

[utilities] [Functions]

DESCRIPTION:

 Get the interpolated value at a specified point within an envelope. The
 envelope must be specified in the form of a list of break-point pairs.

ARGUMENTS:

 - A number that is the point within the specified envelope for which to
   return the interpolated value.
 - A list of break-point pairs.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :scaler. A number that is the factor by which to scale the values of
   the break-point pairs in the given envelope before retrieving the
   interpolated value. Default = 1.
 - :exp. A number that is the exponent to which the result should be
   raised. Default = 1.
 - :warn. T or NIL to indicate whether the method should print a warning if
   the specified point is outside of the bounds of the x-axis specified in
   the list of break-point pairs. T = warn. Default = T.

RETURN VALUE: EXAMPLE:

```
;;; Using the defaults
(interpolate 50 '(0 0 100 1))

=> 0.5

;;; Specifying a different scaler
(interpolate 50 '(0 0 100 1) :scaler 2)

=> 1.0

;;; Specifying a different exponent by which the result is to be raised
(interpolate 50 '(0 0 100 1) :exp 2)

=> 0.25
```

SYNOPSIS:

```
(defun interpolate (point env &key (scaler 1) (exp 1) (warn t))
```

21.34 utilities/invert-env

[utilities] [Functions]

DATE:

 June 15th 2017, Edinburgh

DESCRIPTION:

 Invert an envelope so that its maximum value becomes its minimum,
 vice-versa, and everything inbetween.

ARGUMENTS:

 A list of X-Y breakpoint pairs

RETURN VALUE:

 A list of X-Y breakpoint pairs exhibiting the inversion.

EXAMPLE:

```
(invert-env '(0 0 100 1)) -> (0 1.0 100 0.0)
(invert-env '(0 .3 40 .4 100 .9)) -> (0 0.9 40 0.79999995 100 0.3)
(invert-env '(0 -.9 40 .4 100 .9)) -> (0 0.9 40 -0.39999998 100 -0.9)
```

SYNOPSIS:

```
(defun invert-env (env)
```

21.35 utilities/list-to-string

[utilities] [Functions]

DESCRIPTION:

 Convert a list to a string.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

  - A string that will serve as a separator between the elements.
    Default = " ".
  - T or NIL to indicate whether a list value of NIL is to be returned as
    "NIL" or NIL. T = "NIL" as a string. Default = T.

RETURN VALUE: EXAMPLE:

```
;;; Using defaults
(list-to-string '(1 2 3 4 5))

=> "1 2 3 4 5"

;;; Specifying a different separator
(list-to-string '(1 2 3 4 5) "-")

=> "1-2-3-4-5"

;;; A NIL list returns "NIL" as a string by default
(list-to-string NIL)

=> "nil"

;;; Setting the second optional argument to NIL returns a NIL list as NIL
;;; rather than as "NIL" as a string
(list-to-string NIL "" nil)

=> NIL
```

SYNOPSIS:

```
(defun list-to-string (list &optional (separator " ") (nil-as-string t))
```

21.36   utilities/logarithmic-steps

[utilities] [Functions]

DESCRIPTION:

 Create a list of numbers progressing from the first specified argument to
 the second specified argument over the specified number of steps using an
 exponential curve rather than linear interpolation.

ARGUMENTS:

 - A number that is the starting value in the resulting list.

- A number that is the ending value in the resulting list.
- An integer that will be the length of the resulting list - 1.

OPTIONAL ARGUMENTS:

- A number that will be used as the exponent when determining the
  exponential interpolation between values. Default = 2.

RETURN VALUE:

 A list of numbers.

EXAMPLE:

```
(logarithmic-steps 1 100 19)
```

```
=> (1.0 1.3055556 2.2222223 3.75 5.888889 8.638889 12.0 15.972222 20.555555
    25.75 31.555555 37.97222 45.0 52.63889 60.88889 69.75 79.22222 89.30556
    100.0)
```

SYNOPSIS:

```
(defun logarithmic-steps (low high num-steps &optional (exponent 2))
```

21.37   utilities/middle

[utilities] [Functions]

DESCRIPTION:

 Get the number value that is middle of two number values.

ARGUMENTS:

- A first number.
- A second number.

RETURN VALUE:

 A number.

EXAMPLE:

```
(middle 7 92)
```

```
=> 49.5
```

SYNOPSIS:

```
(defun middle (lower upper)
```

21.38   utilities/mins-secs-to-secs

[utilities] [Functions]

DESCRIPTION:

Derive the number of seconds from a minutes-seconds value that is indicated
as a string of the form "0:00.000" or a two-item list in the form '(minutes
seconds) or three-item list in the form '(minutes seconds milliseconds)

ARGUMENTS:

- A time in minutes and seconds, as described above.

OPTIONAL ARGUMENTS:

- if a string is to be passed, then a character that denotes the separator
between minutes and seconds. Default = #\:

RETURN VALUE:

A decimal number that is a number in seconds.

EXAMPLE:

```
(mins-secs-to-secs '(2 1))
=> 121.0
(mins-secs-to-secs '(16 59 534)))
=> 1019.534
(mins-secs-to-secs "3:06.829"))
=> 186.829
;; using a different separator character between minutes and seconds
(mins-secs-to-secs "3-36.29" #\-) 0.0001)
=> 216.29
```

SYNOPSIS:

```
(defun mins-secs-to-secs (time &optional (post-mins #\:))
```

21.39   utilities/move-elements

[utilities] [Functions]

DATE:

 02-Mar-2011

DESCRIPTION:

 Move the specified elements from one list (if they are present in that
 list) to another, deleting them from the first.

ARGUMENTS:

 - A list of elements that are the elements to be moved.
 - A list from which the specified elements are to be moved and deleted.
 - A list to which the specified elements are to be moved.

OPTIONAL ARGUMENTS:

 - A predicate by which to test that the specified elements are equal to
   elements of the source list. Default = #'eq.

RETURN VALUE:

 Two values: A first list that is the source list after the items have been
 moved; a second list that is the target list after the items have been
 moved.

EXAMPLE:

```
(move-elements '(3 5 8) '(1 2 3 4 5 6 7 8 9) '(a b c d e))

=> (1 2 4 6 7 9), (8 5 3 A B C D E)
```

SYNOPSIS:

```
(defun move-elements (what from to &optional (test #'eq))
```

21.40   utilities/move-to-end

[utilities] [Functions]

DATE:

22-May-2011

DESCRIPTION:

Move a specified element of a given list to the end of the list, returning
the new list.

NB: If the element exists more than once in the given list, all but one of
    the occurrences will be removed and only one of them will be placed at
    the end.

ARGUMENTS:

- An item that is an element of the list that is the second argument.
- A list.

RETURN VALUE:

A list.

EXAMPLE:

```
;;; All unique items
(move-to-end 2 '(1 2 3 4 5))

=> (1 3 4 5 2)

;;; Duplicate items
(move-to-end 2 '(1 2 3 2 4 2 5))

=> (1 3 4 5 2)
```

SYNOPSIS:

```
(defun move-to-end (what list &optional (test #'eql))
```

21.41   utilities/nconc-sublists

[utilities] [Functions]

DESCRIPTION:

Concatenate corresponding sublists of a given list. Each sublist in the
argument should have the same length and number of sublists etc.

ARGUMENTS:

 A list of lists.

RETURN VALUE:

 A list of lists.

EXAMPLE:

```
(nconc-sublists '(((1 2) (a b) (cat dog))
                  ((3 4) (c d) (bird fish))
                  ((5 6) (e f) (pig cow))))

=> ((1 2 3 4 5 6) (A B C D E F) (CAT DOG BIRD FISH PIG COW))
```

SYNOPSIS:

```
(defun nconc-sublists (lists)
```

21.42   utilities/nearest

[utilities] [Functions]

DATE:

 15th May 2020, Heidhausen

DESCRIPTION:

 Return the nearest number in a list to the first argument

ARGUMENTS:

 - the number we're looking to get the closest to
 - the list of numbers we'll search

RETURN VALUE:

 the element of the list that's closest to the first argument, the list
 sorted by nearest to the number, the distances to the number for the sorted
 list.

OPTIONAL ARGUMENTS:

```
none
```

EXAMPLE:

```
(nearest 1.21 '(4 2 5 3 5 4 1.2 1.3 1.1999))
--> 1.2
```

SYNOPSIS:

```
(defun nearest (num list)
```

21.43   utilities/nearest-power-of-2

[utilities] [Functions]

DESCRIPTION:

Return the closest number to the specified value that is a power of two but
not greater than the specified value (unless the optional argument is T:
see below).

ARGUMENTS:

- A number.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether we can return a power of 2 greater than the
  argument, if that is nearer to the argument than the lower power or 2.

RETURN VALUE:

An integer that is a power of two.

EXAMPLE:

```
(nearest-power-of-2 31)
```

```
=> 16
```

```
(nearest-power-of-2 31 t)
```

```
=> 32
```

```
(nearest-power-of-2 32)
```

```
=> 32
```

```
(nearest-power-of-2 33)
```

```
=> 32
```

SYNOPSIS:

```
(defun nearest-power-of-2 (num &optional allow>)
```

21.44   utilities/now-string

[utilities] [Functions]

DATE:

```
 Sat 28 Mar 2020 13:21:08 GMT - London
```

DESCRIPTION:

```
 Return a string representing the current time in the format:
 YEAR MONTH DAY - HOURS MINUTES SECONDS
 e.g. "20200328-132227"

 It is thought that this function might be useful when outputing multiple
 files during the test phase of a piece. E.g.
 (cmn-display +mini+
              :file (concatenate 'string "my-piece" (now-string) ".eps"))
```

ARGUMENTS:

```
 None
```

OPTIONAL ARGUMENTS:

```
 None
```

RETURN VALUE:

```
 A string
```

EXAMPLE:

```
(now-string)
```

```
=> "20200328-132227"
```

```
(concatenate 'string "my-piece_" (now-string) ".eps")
```

```
=> "my-piece_20200328-133357.eps"
```

SYNOPSIS:

```
(defun now-string ()
```

21.45   utilities/octave-freqs

[utilities] [Functions]

DESCRIPTION:

A boolean test to determine whether two specified frequencies are octave
transpositions of the same pitch class.

ARGUMENTS:

- A first number that is a frequency in Hertz.
- A second number that is a frequency in Hertz.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether identical frequencies ("unison") are also
  to be considered octave transpositions of the same pitch class.
  T = unisons are also octaves. Default = T.

RETURN VALUE:

T or NIL.

EXAMPLE:

```
(octave-freqs 261.63 2093.04)
```

```
=> T
```

```
(octave-freqs 261.63 3000.00)
```

```
=> NIL
```

```
(octave-freqs 261.63 261.63)
```

```
=> T
```

```
(octave-freqs 261.63 261.63 nil)
```

```
=> NIL
```

SYNOPSIS:

```
(defun octave-freqs (freq1 freq2 &optional (unison-also t))
```

21.46   utilities/one-to-many

[utilities] [Functions]

DATE:

 June 23rd 2020

DESCRIPTION:

 Find a one-to-many relationship between the first argument and a number of
 equally-spaced points given in the second argument. The first argument is a
 number between 0.0 and 1.0 (inclusive). We calculate the proximities from
 this point to the number of points given and return them as a list,
 optionally raised to a given exponent. The list returned is scaled so that
 all values sum to 1.0, so this is particularly useful for, say, calculating
 a number of amplitude scalers for a multi-voice synthesis process.

ARGUMENTS:

 - the point: a number between 0.0 and 1.0 inclusive
 - the number of points to use in the calculation. This will also be the
   number of results returned. Alternatively this can be a list of numbers
   between 0.0 and 1.0. This way you can pass your own points for e.g. an
   unequally-spaced set. In this case though the proximity is still
   determined from a maximum of 1.0, not the highest number in the given
   list.

OPTIONAL ARGUMENTS:

 - the exponent to raise proximities to. 1.0 will return a linear

relationship. > 1.0 will exaggerate the relationships so that those points
further away from the first argument will be pushed further away than a
linear relationship. < 1.0 will lessen the distances.

RETURN VALUE:

 a list of numbers the length of which is the same as the 2nd argument and
 the sum of which is 1.0

EXAMPLE:

```
(one-to-many .8 7) -->
(0.045112778 0.082706764 0.12030074 0.15789473 0.19548872 0.21804512 0.18045112)
(one-to-many .8 7 .7) -->
(0.06509622 0.09950039 0.1293403 0.15646003 0.18169008 0.19612299 0.17178996)
(one-to-many .8 7 1.3)  -->
(0.030845987 0.06782856 0.11039718 0.15721251 0.20752355 0.23917702 0.1870151)
;;; passing 5 points: these don't have to have min/max of 0 and 1 ...
(one-to-many .8 '(0 .1 .35 .7 .92)) -->
(0.07067137 0.10600708 0.19434628 0.3180212 0.31095406)
;;; ... and they don't have to be in ascending order either
(one-to-many .8 '(0 .1 .35 .7 .2)) -->
(0.08510638 0.12765959 0.23404254 0.38297874 0.17021276)
```

SYNOPSIS:

```
(defun one-to-many (one how-many &optional (expt 1.0))
```

21.47   utilities/parse-audacity-label-file

[utilities] [Functions]

DATE:

 August 12th 2019

DESCRIPTION:

 Parse a labels file exported from Audacity and return a list of (start end
 label) triplets. Times are in seconds.

ARGUMENTS:

 - the path to the label file

RETURN VALUE:

 a list

SYNOPSIS:

(defun parse-audacity-label-file (label-file)

21.48   utilities/parse-audacity-label-file-for-loops

[utilities] [Functions]

DESCRIPTION:

 Read an audacity label file and return its loop points as groups.

 NB: If this fails it's probably because there's a tab between time and
     label instead of spaces: save in emacs to detab.

 NB: Beware that marker files created on different operating systems from
     the one on which this function is called might trigger errors due to
     newline character mismatches.

ARGUMENTS:

 - A string that is the name of the label file to be parsed, including
   directory path and extension.

RETURN VALUE:

 Returns a list of lists which are the grouped time points.

 Also prints separate feedback to the listener.

EXAMPLE:

(parse-audacity-label-file-for-loops "/path/to/24-7loops1.txt")

=>
313 markers, 50 loops read

((25.674559 25.829296 26.116327 26.649048 27.038843)
 (32.211884 32.33669 32.481815 32.618233 32.716915 32.902676 33.227757
  33.61959)

```
(36.893604 37.059048 37.160633 37.27383 37.439274 37.4683 37.627937)
(39.52907 39.81932 39.999275 40.2634 40.338867 40.605896)
(45.612698 45.818775 46.050976 46.145306 46.275192)
(46.4566 46.644535 46.76934 46.886894 46.971066 47.16553)
(84.15927 84.260864 84.292786 84.355194 84.47274 84.52789 84.556915
 84.65415)
...
(676.1075 676.79114 677.1503 677.57904 678.12366)
(799.29205 799.8019 800.58984 800.96063 801.13446 801.45886)
(804.98145 805.2016 805.5724 805.83887 806.31396))
```

SYNOPSIS:

```
(defun parse-audacity-label-file-for-loops (label-file)
```

21.49   utilities/parse-wavelab-marker-file-for-loops

[utilities] [Functions]

DESCRIPTION:

```
Read a wavelab marker file and return its loop points as groups.

The marker file must contain markers with the word "loop". A marker with
that name will start a new set of loop points, and nameless markers will
belong to the group until the next "loop" marker.
```

ARGUMENTS:

```
- A string that is the name of the marker file to be parsed, including
  directory path and extension.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :sampling-rate. An integer that is the sampling rate of the sound file to
  which the marker file refers. This value will affect the resulting time
  points. Default = 44100.
- :max-length. The maximum duration in seconds between two points: anything
  greater than this will result in a warning being printed.
```

RETURN VALUE:

```
Returns a list of lists which are the grouped time points.

Also prints separate feedback to the listener.
```

EXAMPLE:

```
(parse-wavelab-marker-file-for-loops "/path/to/24-7loops1.mrk")

=>
WARNING:
   utilities::parse-wavelab-marker-file-for-loops
   loop points 10:13.213 to 10:14.475 are too long (1.2620239)
WARNING:
   utilities::parse-wavelab-marker-file-for-loops
   loop points 10:33.223 to 10:34.486 are too long (1.2630615)
WARNING:
   utilities::parse-wavelab-marker-file-for-loops
   loop points 10:36.456 to 10:37.522 are too long (1.06604)


312 markers, 50 loops read

((25.674559 25.829296 26.116327 26.649048 27.038843)
 (32.211884 32.33669 32.481815 32.618233 32.716915 32.902676 33.227757
  33.61959)
 (36.893604 37.059048 37.160633 37.27383 37.439274 37.4683 37.627937)
 (39.52907 39.81932 39.999275 40.2634 40.338867 40.605896)
 (45.612698 45.818775 46.050976 46.145306 46.275192)
 (46.4566 46.644535 46.76934 46.886894 46.971066 47.16553)
 (84.15927 84.260864 84.292786 84.355194 84.47274 84.52789 84.556915
  84.65415)
 ...
 (655.91077 656.4554 656.80304 657.4519 658.04285 658.8192)
 (676.1075 676.79114 677.1503 677.57904 678.12366)
 (799.29205 799.8019 800.58984 800.96063 801.13446 801.45886)
 (804.98145 805.2016 805.5724 805.83887 806.31396))
```

SYNOPSIS:

```
(defun parse-wavelab-marker-file-for-loops
    (marker-file &key (sampling-rate 44100) (max-length 1.0))
```

21.50   utilities/partial-freqs

[utilities] [Functions]

DATE:

 13-Dec-2011

DESCRIPTION:

 A Boolean test to determine whether either of two specified frequencies
 can be considered a harmonic partial of the other.

ARGUMENTS:

 - A first frequency in Hertz.
 - A second frequency in Hertz.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether identical frequencies ("unison") are also to
   be considered partials of each other. T = unison are partials.
   Default = T.

RETURN VALUE:

 T if one of the frequencies has the ratio of a harmonic partial to the
 other, otherwise NIL.

EXAMPLE:

```
(partial-freqs 300 900)

=> T

(partial-freqs 300 700)

=> NIL

(partial-freqs 300 300)

=> T

(partial-freqs 300 300 nil)

=> NIL
```

SYNOPSIS:

```
(defun partial-freqs (freq1 freq2 &optional (unison-also t))
```

21.51   utilities/pdivide

[utilities] [Functions]

DESCRIPTION:

Creates a list of proportionally related times, dividing a starting
duration into a number of smaller durations a specified number of times.
We start with a proportion as a ratio (e.g. 3/2) and divide the given
duration into two parts according to that ratio. Then those two parts will
be divided into the same ratios. This will iterate the number of times
indicated by the second argument.

The following are some classical proportions:
```
        Latin          (Greek)
(3 : 2) Sesquialtera (Diapente)
(4 : 3) Sesquitertia (Diatessaron)
(5 : 4) Sesquiquarta (Diatonus Semitonus)
(8 : 3) Duplasuperbipartiens (Diapson Diatesseron)
(9 : 8) Sesquioctava (Tonus)
```

ARGUMENTS:

- an integer or ratio (in Lisp terms, a rational) e.g. 3/2
- an integer >=1 specifying the number of times to iterate the process of
  dividing the duration into proportions.

OPTIONAL ARGUMENTS:

keyword arguments:
- :duration. The overall duration to apply the proportional divisions to.
  Units are arbitrary of course as this is just a number. Default 1.0.
- :print. If T, print each level of division as we proceed. Default NIL.
- :reverse. If T reverse the proportion (so 3/2 becomes 2/3). Default NIL.
- :alternate. If T, reverse the proportion every other division (not
  iteration) so that if we have a proportion of 3/2 on the second iteration
  we divide into 3/2 then 2/3. Default NIL.
- :increment. If T, then each time we divide we increment both sides of the
  proportion, so 3:2 becomes 4:3 which becomes 5:4 etc. Default NIL.
- :halves. This will only make a difference if :increment is T: As results
  tend overall towards increasing (when numerator < denominator e.g. 2/3) or
  decreasing (numerator > denominator e.g. 3/2) numbers, we can mix things
  up by dividing the resultant list into two halves and splicing their
  elements one after the other. Default NIL.
- :shuffle. Mix things up by shuffling the resultant list. As this uses
  the shuffle algorithm we have fixed-seed randomness so results will be

the same upon each call within the same Lisp implementation/version.
Default NIL.

RETURN VALUE:

Three values: the list of ascending timings from the last generation of the
calculated proportions; the durations of each part for the last generation;
the list of ascending timings for _each_ generation of the calculated
proportions (a list of lists).

EXAMPLE:

Notice here that each generation prints the proportions along with the
durations these correspond to and the start time of each (cumulative durations).

```
(pdivide 3/2 4 :duration 35 :print t)
```

```
PRINTS:
Generation 1: 3 (21.00=21.00), 2 (14.00=35.00),

Generation 2: 3 (12.60=12.60), 2 (8.40=21.00), 3 (8.40=29.40), 2 (5.60=35.00),

Generation 3: 3 (7.56=7.56), 2 (5.04=12.60), 3 (5.04=17.64), 2 (3.36=21.00),
3 (5.04=26.04), 2 (3.36=29.40), 3 (3.36=32.76), 2 (2.24=35.00),

Generation 4: 3 (4.54=4.54), 2 (3.02=7.56), 3 (3.02=10.58), 2 (2.02=12.60),
3 (3.02=15.62), 2 (2.02=17.64), 3 (2.02=19.66), 2 (1.34=21.00), 3 (3.02=24.02),
2 (2.02=26.04), 3 (2.02=28.06), 2 (1.34=29.40), 3 (2.02=31.42), 2 (1.34=32.76),
3 (1.34=34.10), 2 (0.90=35.00),

RETURNS:
(0.0 4.5360003 7.5600004 10.584001 12.6 15.624001 17.640001 19.656002 21.000002
 24.024002 26.040003 28.056004 29.400003 31.416004 32.760006 34.104008
 35.000008)
(4.5360003 3.0240002 3.0240004 2.0160003 3.0240004 2.0160003 2.0160003
 1.3440002 3.0240004 2.0160003 2.0160003 1.3440002 2.0160003 1.3440001
 1.3440001 0.896)
((0.0 4.5360003 7.5600004 10.584001 12.6 15.624001 17.640001 19.656002
  21.000002 24.024002 26.040003 28.056004 29.400003 31.416004 32.760006
  34.104008 35.000008)
 (0.0 7.5600004 12.6 17.640001 21.000002 26.040003 29.400003 32.760002
  35.000004)
 (0.0 12.6 21.0 29.400002 35.0) (0.0 21.0 35.0))
```

```
(pdivide 3/2 4 :duration 35 :print t :increment t :halves t)
```

```
PRINTS:
Generation 1: 3 (21.00=21.00), 2 (14.00=35.00),

Generation 2: 4 (12.00=12.00), 3 (9.00=21.00), 5 (7.78=28.78), 4 (6.22=35.00),

Generation 3: 6 (6.55=6.55), 5 (5.45=12.00), 7 (4.85=16.85), 6 (4.15=21.00),
8 (4.15=25.15), 7 (3.63=28.78), 9 (3.29=32.07), 8 (2.93=35.00),

Generation 4: 10 (3.44=3.44), 9 (3.10=6.55), 11 (2.86=9.40), 10 (2.60=12.00),
12 (2.53=14.53), 11 (2.32=16.85), 13 (2.16=19.01), 12 (1.99=21.00),
14 (2.15=23.15), 13 (2.00=25.15), 15 (1.88=27.03), 14 (1.75=28.78),
16 (1.70=30.48), 15 (1.59=32.07), 17 (1.51=33.58), 16 (1.42=35.00),

RETURNS:
(0.0 3.4449766 5.595868 8.696347 10.6936035 13.550747 15.428142 18.025545
 19.77778 22.30621 24.0064 26.324125 27.918053 30.078053 31.58647 33.580315
 35.0)
(3.4449766 2.1508918 3.100479 1.9972568 2.8571434 1.8773947 2.5974028 1.752235
 2.5284283 1.7001898 2.317726 1.593928 2.16 1.5084175 1.9938462 1.4196872)
((0.0 3.4449766 6.5454555 9.402599 12.000002 14.52843 16.846155 19.006155
  21.000002 23.150894 25.148151 27.025547 28.777782 30.477972 32.0719 33.58032
  35.000004)
 (0.0 6.5454555 12.000002 16.846157 21.000004 25.148151 28.77778 32.0719
  35.000004)
 (0.0 12.000001 21.0 28.777779 35.0) (0.0 21.0 35.0))
```

SYNOPSIS:

```
(defun pdivide (start levels &key (duration 1.0) print reverse alternate
                       halves shuffle increment)
```

21.52   utilities/pexpand

[utilities] [Functions]

DESCRIPTION:

 Instead of dividing an overall duration (pdivide) we start with a
 proportion and expand outwards from there, keeping each newly created part
 in the same proportion. This is repeated the number of times specified in
 the first argument. Useful for generating maps (section structure).

ARGUMENTS:

 The number of times to expand proportionally.

OPTIONAL ARGUMENTS:

 As many integer proportions as required. If the last argument here is t,
 then instead of using letters to denote sections we use numbers instead.

RETURN VALUE:

 3 values:
 1) a list showing the cumulative count (e.g. bar numbers) of where major
 and minor sections occur. Topmost sections will have the labels A, B, C,
 etc. with subsections such as A.A, A.B, ... C.C.C.C. Of course, wherever a
 major section starts, an arbitrary number of subsections also begin, but
 only the most major section is present in the list.
 2) the structure of the sections and subsections in the form of a list of
 sublists for each, and containing the section labels paired with their
 length. The bottommost subsection will have a length of the sum of the
 proportions, with higher subsection groupings showing multiples of this.
 3) the overall length of the structure produced (also the first element of
 the second returned value).

EXAMPLE:

```
;;; 2 generations:
(pexpand 2 3 2) =>
(1 (A) 6 (A A A B) 11 (A A A C) 16 (A A B) 21 (A A B B) 26 (A B) 31 (A B A B)
 36 (A B A C) 41 (A B B) 46 (A B B B) 51 (A C) 56 (A C A B) 61 (A C A C) 66
 (A C B) 71 (A C B B) 76 (B) 81 (B A A B) 86 (B A A C) 91 (B A B) 96 (B A B B)
 101 (B B) 106 (B B A B) 111 (B B A C) 116 (B B B) 121 (B B B B))
(125
 (((A) 75)
  (((A A) 25) (((A A A) 15) ((A A A A) 5) ((A A A B) 5) ((A A A C) 5))
   (((A A B) 10) ((A A B A) 5) ((A A B B) 5)))
  (((A B) 25) (((A B A) 15) ((A B A A) 5) ((A B A B) 5) ((A B A C) 5))
   (((A B B) 10) ((A B B A) 5) ((A B B B) 5)))
  (((A C) 25) (((A C A) 15) ((A C A A) 5) ((A C A B) 5) ((A C A C) 5))
   (((A C B) 10) ((A C B A) 5) ((A C B B) 5))))
 (((B) 50)
  (((B A) 25) (((B A A) 15) ((B A A A) 5) ((B A A B) 5) ((B A A C) 5))
   (((B A B) 10) ((B A B A) 5) ((B A B B) 5)))
  (((B B) 25) (((B B A) 15) ((B B A A) 5) ((B B A B) 5) ((B B A C) 5))
   (((B B B) 10) ((B B B A) 5) ((B B B B) 5)))))
125

;;; 3 generations:
(pexpand 3 3 2) =>
```

```
(1 (A) 6 (A A A A A B) 11 (A A A A A C) 16 (A A A A B) 21 (A A A A B B) 26
 (A A A B) 31 (A A A B A B) 36 (A A A B A C) 41 (A A A B B) 46 (A A A B B B) 51
 (A A A C) 56 (A A A C A B) 61 (A A A C A C) 66 (A A A C B) 71 (A A A C B B) 76
...
 581 (B B B A A B) 586 (B B B A A C) 591 (B B B A B) 596 (B B B A B B) 601
 (B B B B) 606 (B B B B A B) 611 (B B B B A C) 616 (B B B B B) 621
 (B B B B B B))
(625
 (((A) 375)
  (((A A) 125)
   (((A A A) 75)
    (((A A A A) 25)
     (((A A A A A) 15) ((A A A A A A) 5) ((A A A A A B) 5) ((A A A A A C) 5))
     (((A A A A B) 10) ((A A A A B A) 5) ((A A A A B B) 5)))
...
   (((B B B) 50)
    (((B B B A) 25)
     (((B B B A A) 15) ((B B B A A A) 5) ((B B B A A B) 5) ((B B B A A C) 5))
     (((B B B A B) 10) ((B B B A B A) 5) ((B B B A B B) 5)))
    (((B B B B) 25)
     (((B B B B A) 15) ((B B B B A A) 5) ((B B B B A B) 5) ((B B B B A C) 5))
     (((B B B B B) 10) ((B B B B B A) 5) ((B B B B B B) 5)))))))
625

;;; 2 generations of 3 proportional values, returning numbers for labels
(pexpand 2 3 2 4 t) =>
(1 (1) 10 (1 1 1 2) 19 (1 1 1 3) 28 (1 1 2) 37 (1 1 2 2) 46 (1 1 3) 55
 (1 1 3 2) 64 (1 1 3 3) 73 (1 1 3 4) 82 (1 2) 91 (1 2 1 2) 100 (1 2 1 3) 109
 (1 2 2) 118 (1 2 2 2) 127 (1 2 3) 136 (1 2 3 2) 145 (1 2 3 3) 154 (1 2 3 4)
... (3 4 2 2) 694 (3 4 3) 703 (3 4 3 2) 712 (3 4 3 3) 721 (3 4 3 4))

(729
 (((1) 243)
  (((1 1) 81) (((1 1 1) 27) ((1 1 1 1) 9) ((1 1 1 2) 9) ((1 1 1 3) 9))
   (((1 1 2) 18) ((1 1 2 1) 9) ((1 1 2 2) 9))
   (((1 1 3) 36) ((1 1 3 1) 9) ((1 1 3 2) 9) ((1 1 3 3) 9) ((1 1 3 4) 9)))
...
   (((3 2 2) 18) ((3 2 2 1) 9) ((3 2 2 2) 9))
   (((3 2 3) 36) ((3 2 3 1) 9) ((3 2 3 2) 9) ((3 2 3 3) 9) ((3 2 3 4) 9)))
  (((3 3) 81) (((3 3 1) 27) ((3 3 1 1) 9) ((3 3 1 2) 9) ((3 3 1 3) 9))
   (((3 3 2) 18) ((3 3 2 1) 9) ((3 3 2 2) 9))
   (((3 3 3) 36) ((3 3 3 1) 9) ((3 3 3 2) 9) ((3 3 3 3) 9) ((3 3 3 4) 9)))
  (((3 4) 81) (((3 4 1) 27) ((3 4 1 1) 9) ((3 4 1 2) 9) ((3 4 1 3) 9))
   (((3 4 2) 18) ((3 4 2 1) 9) ((3 4 2 2) 9))
   (((3 4 3) 36) ((3 4 3 1) 9) ((3 4 3 2) 9) ((3 4 3 3) 9) ((3 4 3 4) 9)))))
729
```

SYNOPSIS:

```
(defun pexpand (generations &rest proportions)
```

21.53   utilities/pexpand-find

[utilities] [Functions]

DESCRIPTION:

Find the cumulative number of where a label occurs in a list returned by
pexpand.

ARGUMENTS:

- the label we're looking for
- a list of the type returned by pexpand (first returned value).

OPTIONAL ARGUMENTS:

- a function to be called when the label cannot be found. Default =
#'error but could also be #'warn or NIL.

RETURN VALUE:

An integer.

SYNOPSIS:

```
(defun pexpand-find (label list &optional (on-error #'error))
```

21.54   utilities/pexpand-reaper-markers

[utilities] [Functions]

DATE:

September 4th 2015, Edinburgh

DESCRIPTION:

Using the pexpand function, we write marker information in a format which
can be read by the Reaper (version 4/5+) DAW software. Though we can think
of the outputs of pexpand to be in beats, seconds, bars, or any arbitrary

scale, the timings of Reaper markers are in seconds, hence the need here
for an initial tempo. In other words, we treat the output of pexpand to be
beat counts; if you would prefer to interpret these as bars, simply divide
the tempo by the number of beats per bar. If there are to be tempo changes
in the mix/piece Reaper itself will update the markers' positions when the
new tempo is inserted in the project window--this is fine when you are
thinking in beats/bars but beware of changing tempo in Reaper if you are
thinking of markers with fixed timings (seconds).

Copy the output of this into the Reaper file verbatim (no enclosing < >
marks) before the <PROJBAY tag.

ARGUMENTS:

 - the tempo in BPM
 - the number of generations: see the pexpand function
 - (&rest) the proportions: see the pexpand function

RETURN VALUE:

 Always T

EXAMPLE:

```
(pexpand-reaper-markers 144 2 6 3 5 4)
->
  MARKER 1 7.5 "level 4" 0 0 1
  MARKER 2 15.0 "level 4" 0 0 1
  MARKER 3 22.5 "level 4" 0 0 1
  MARKER 4 30.0 "level 4" 0 0 1
  MARKER 5 37.5 "level 4" 0 0 1
...
  MARKER 108 810.0 "level 1" 0 0 1
  MARKER 109 817.5 "level 4" 0 0 1
  MARKER 110 825.0 "level 4" 0 0 1
  MARKER 111 832.5 "level 4" 0 0 1
  MARKER 112 840.0 "level 4" 0 0 1
  MARKER 113 847.5 "level 4" 0 0 1
  MARKER 114 855.0 "level 3" 0 0 1
...
  MARKER 319 2392.5 "level 4" 0 0 1
  MARKER 320 2400.0 "level 3" 0 0 1
  MARKER 321 2407.5 "level 4" 0 0 1
  MARKER 322 2415.0 "level 4" 0 0 1
  MARKER 323 2422.5 "level 4" 0 0 1
```

Here's where I pasted the data into the .RPP Reaper file:

```
  <TEMPOENVEX
    ACT 0
    VIS 1 0 1
    LANEHEIGHT 0 0
    ARM 0
    DEFSHAPE 1 -1 -1
  >
  MARKER 1 7.5 "level 4" 0 0 1
  MARKER 2 15 "level 4" 0 0 1
  MARKER 3 22.5 "level 4" 0 0 1
  MARKER 4 30 "level 4" 0 0 1
...
  MARKER 320 2400 "level 3" 0 0 1
  MARKER 321 2407.5 "level 4" 0 0 1
  MARKER 322 2415 "level 4" 0 0 1
  MARKER 323 2422.5 "level 4" 0 0 1
  <PROJBAY
  >
  <TRACK {EBF9837F-BE25-9542-B720-A1862C0DF380}
```

SYNOPSIS:

```
(defun pexpand-reaper-markers (tempo generations &rest proportions)
```

21.55   utilities/pexpand-section-length

[utilities] [Functions]

DESCRIPTION:

Return the length (integer) of any arbitrary section in the data returned
by pexpand.

ARGUMENTS:

- The (rest of) the kind of list returned as the second value of a call to
  pexpand.
- The section ID we want the length of, either as a list or single symbol.

RETURN VALUE:

 An integer or NIL if the section can't be found.

EXAMPLE:

```
(pexpand-section-length (rest (nth-value 1 (pexpand 2 3 6 4 5))) '(c a b))
=> 108

(pexpand-section-length (rest (nth-value 1 (pexpand 2 3 6 4 5))) 'c)
=> 1296
```

SYNOPSIS:

```
(defun pexpand-section-length (pexpand-list section)
```

21.56 utilities/power-of-2

[utilities] [Functions]

DESCRIPTION:

 Test whether the specified number is a power of two and return the
 logarithm of the specified number to base 2.

 This method returns two values: T or NIL for the test and a decimal that is
 the logarithm of the specified number to base 2.

ARGUMENTS:

 - A number.

RETURN VALUE:

 Two values: T or NIL for the test and a decimal number that is the
 logarithm of the specified number to base 2.

EXAMPLE:

```
(power-of-2 16)
```

```
=> T, 4.0
```

```
(power-of-2 17.3)
```

```
=> NIL, 4.1127
```

SYNOPSIS:

```
(defun power-of-2 (float)
```

21.57   utilities/prime

[utilities] [Functions]

DATE:

 August 5th 2019, Heidhausen

DESCRIPTION:

 Taken from dlocsig.lisp by Fernando Lopez Lezcano (in the CLM package):
 Return T or NIL to indicated whether the argument is a prime number or not.

ARGUMENTS:

 an integer (all other types, including floats, will trigger an error)

RETURN VALUE:

 T or NIL

SYNOPSIS:

(defun prime (val)

21.58   utilities/pts2cm

[utilities] [Functions]

DESCRIPTION:

 Convert a specified number of points to a length in centimeters at a
 resolution of 72ppi.

ARGUMENTS:

 - A number.

RETURN VALUE:

 A number.

EXAMPLE:

```
(pts2cm 150)
```

```
=> 5.2916665
```

SYNOPSIS:

```
(defun pts2cm (points)
```

21.59   utilities/random-amount

[utilities] [Functions]

DESCRIPTION:

```
Return a random number from within a total range of <percent> of the given
number, centering around zero. Thus, if the <number> is 100, and the
<percent> is 5, the results will be a random number between -2.5 and +2.5.
```

ARGUMENTS:

```
A number.
```

OPTIONAL ARGUMENTS:

```
A number that will be a percent of the given number.
```

RETURN VALUE:

```
A random positive or negative number.
```

EXAMPLE:

```
;;; Using the default will return numbers within a 5% span of the given number,
;;; centering around zero. With 100 that means between -2.5 and +2.5.
(loop repeat 10 collect (random-amount 100))
```

```
=> (0.7424975 -1.4954442 -1.7126495 1.5918689 -0.43478793 -1.7916341 -1.9115914
    0.8541988 0.057197176 2.0713913)
```

```
;;; Specifying 10% of 80 will return random numbers between -4.0 and +4.0
(loop repeat 10 collect (random-amount 80 10))
```

```
=> (-0.66686153 3.0387697 3.4737322 -2.3753185 -0.8495751 -0.47580242
    -0.25743783 -1.1395472 1.3560238 -0.5958566)
```

SYNOPSIS:

```
(defun random-amount (number &optional (percent 5))
```

21.60 utilities/random-from-list

[utilities] [Functions]

DESCRIPTION:

 Return a random element from a specified list of elements.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

 - An integer can be passed stating the length of the list, for more
   efficient processing. NB: There is no check to ensure this number is
   indeed the length of the list. If the number is less than the length of
   the list, only elements from the first part of the list will be
   returned. If it is greater than the length of the list, the method may
   return NIL.

RETURN VALUE:

 An element from the specified list.

EXAMPLE:

(random-from-list '(3 5 7 11 13 17 19 23 29))

=> 13

SYNOPSIS:

(defun random-from-list (list &optional list-length) ; for efficiency

21.61 utilities/randomise

[utilities] [Functions]

DESCRIPTION:

 Return a random decimal number close to (+ or -) the number specified
 (within a certain percentage of that number's value). Note that if you want
 the result to go from 0 to 2x the argument, then <percent> needs to be 200,
 not 100.

ARGUMENTS:

 - A number.

OPTIONAL ARGUMENTS:

 - A number that is a percentage value, such that any random number returned
   will be within that percentage of the original number's value.
   Default = 5.

RETURN VALUE:

 A decimal number.

EXAMPLE:

```
(loop repeat 10 collect (randomise 100))
```

```
=> (99.413795 99.15346 98.682014 100.76199 97.74929 99.05693 100.59494 97.96452
    100.42091 100.01329)
```

SYNOPSIS:

```
(defun randomise (number &optional (percent 5))
```

21.62   utilities/read-from-file

[utilities] [Functions]

DESCRIPTION:

 Read a Lisp expression from a file. This is determined by the Lisp
 parenthetical syntax.

ARGUMENTS:

 - A string that is a file name including directory path and extension.

RETURN VALUE:

 The Lisp expression contained in the file.

EXAMPLE:

```
(read-from-file "/path/to/lisp-lorem-ipsum.txt")
```

```
=> (LOREM IPSUM DOLOR SIT AMET CONSECTETUR ADIPISCING ELIT CRAS CONSEQUAT
      CONVALLIS JUSTO VITAE CONSECTETUR MAURIS IN NIBH VEL EST TEMPUS LOBORTIS
      SUSPENDISSE POTENTI SED MAURIS MASSA ADIPISCING VITAE DIGNISSIM CONDIMENTUM
      VOLUTPAT VEL FELIS FUSCE AUGUE DUI PULVINAR ULTRICIES IMPERDIET SED
      PHARETRA EU QUAM INTEGER IN VULPUTATE VELIT ALIQUAM ERAT VOLUTPAT VIVAMUS
      SIT AMET ORCI EGET EROS CONSEQUAT TINCIDUNT NUNC ELEMENTUM ADIPISCING
      LOBORTIS MORBI AT LOREM EST EGET MATTIS ERAT DONEC AC RISUS A DUI MALESUADA
      LOBORTIS AC AT EST INTEGER AT INTERDUM TORTOR VIVAMUS HENDRERIT CONSEQUAT
      AUGUE QUISQUE ALIQUAM TELLUS NEC VESTIBULUM LOBORTIS RISUS TURPIS LUCTUS
      LIGULA IN BIBENDUM FELIS SEM PULVINAR DOLOR VIVAMUS RHONCUS NISI GRAVIDA
      PORTA VULPUTATE IPSUM LACUS PORTA RISUS A VULPUTATE MAGNA JUSTO A EST)
```

SYNOPSIS:

```
(defun read-from-file (file)
```

21.63   utilities/reflect-list

[utilities] [Functions]

DESCRIPTION:

Order a list of numbers from least to greatest, then transpose the list so
that if an element is the second lowest, it will be replaced by the second
highest etc.

ARGUMENTS:

- A list or numbers.

RETURN VALUE:

A list of numbers.

EXAMPLE:

```
(reflect-list '(1 4 3 5 9 6 2 7 8 8 9))
```

```
=> (9 6 7 5 1 4 8 3 2 2 1)
```

SYNOPSIS:

```
(defun reflect-list (list)
```

21.64   utilities/remove-all

[utilities] [Functions]

DESCRIPTION:

 Remove all of the specified elements from a list, returning a list
 containing only those elements that are not in the first argument list.

ARGUMENTS:

 - A first list that is the list of items to remove.
 - A second list that is the original list.

OPTIONAL ARGUMENTS:

 - A predicate for testing equality between the elements of the two lists.
   Default = #'eq.

RETURN VALUE:

 A list.

EXAMPLE:

```
(remove-all '(3 5 8 13) '(1 2 3 4 5 6 7 8 9 10 11 12 13))

=> (1 2 4 6 7 9 10 11 12)
```

SYNOPSIS:

```
(defun remove-all (rm-list list &optional (test #' eq))
```

21.65   utilities/remove-elements

[utilities] [Functions]

DESCRIPTION:

 Remove a specified number of elements from a given list starting at a
 specified position (0-based) within the list.

ARGUMENTS:

- A list.
- An integer that is the 0-based position within that list that will be the
  first element to be removed.
- An integer that is the number of elements to remove.

RETURN VALUE:

 A list.

EXAMPLE:

(remove-elements '(1 2 3 4 5 6 7) 2 4)

=> (1 2 7)

SYNOPSIS:

(defun remove-elements (list start how-many)

21.66   utilities/remove-more

[utilities] [Functions]

DESCRIPTION:

 Remove all instances of a list of specified elements from an original
 list. The predicate used to test the presence of the specified elements in
 the original list must be specified by the user (such as #'eq, #'equalp,
 #'= etc.)

ARGUMENTS:

 - A list.
 - A predicate with which to test the presence of the specified elements.
 - A sequence of elements to be removed from the given list.

RETURN VALUE:

 A list.

EXAMPLE:

(remove-more '(1 2 3 4 5 5 5 6 7 7 8) #'= 5 7 2)

=> (1 3 4 6 8)

SYNOPSIS:

```
(defun remove-more (list test &rest remove)
```

21.67   utilities/replace-elements

[utilities] [Functions]

DESCRIPTION:

Replace the elements in list between start and end (inclusive) with the new
list.

ARGUMENTS:

- A list.
- An integer that is first position of the segment of the original list to
  be replaced.
- An integer that is the last position of the segment of the original list
  to be replaced.
- A list that is to replace the specified segment of the original
  list. This list can be of a different length than that of the segment
  of the original specified by the start and end positions.

RETURN VALUE:

A list.

EXAMPLE:

```
(replace-elements '(1 2 3 4 5 6 7 8 9) 3 7 '(dog cat goldfish))
```

```
=> (1 2 3 DOG CAT GOLDFISH 9)
```

SYNOPSIS:

```
(defun replace-elements (list start end new)
```

21.68   utilities/rescale

[utilities] [Functions]

DATE:

June 8th 2016, Edinburgh

DESCRIPTION:

 Given a value within an original range, return its value withing a new range

ARGUMENTS:

 - the value we want to rescale
 - the original minimum
 - the original maximum
 - the new minimum
 - the new maximum

OPTIONAL ARGUMENTS:

 keyword arguments:
 :out-of-range. The function to call when the first argument is not within
 the range of arguments two and three. This would normally be #'error,
 #'warn or NIL. If #'warn or NIL, argument 1 will be hard-limited to the
 original range. Default = #'error

RETURN VALUE:

 The value within the new range (a number)

EXAMPLE:

```
(rescale .5 0 1 0 100)
==> 50.0
```

SYNOPSIS:

```
(defun rescale (val min max new-min new-max &optional (out-of-range #'error))
```

21.69   utilities/round-if-close

[utilities] [Functions]

DESCRIPTION:

 Round a decimal number if it is within a given tolerance to the next whole
 number.

ARGUMENTS:

- A decimal number.

OPTIONAL ARGUMENTS:

- If the given number is this amount or less than the nearest whole number,
  round the given number to the nearest whole number.

RETURN VALUE:

If the given number is within the tolerance, return the number, otherwise
return the nearest whole number.

EXAMPLE:

```
(round-if-close 1.999998)
```

```
=> 1.999998
```

```
(round-if-close 1.999999)
```

```
=> 2
```

SYNOPSIS:

```
(defun round-if-close (num &optional (tolerance 0.000001))
```

21.70 utilities/scale-env

[utilities] [Functions]

DESCRIPTION:

Scale either the x-axis values, the data values, or both of a list of
break-point pairs by specified factors.

ARGUMENTS:

- An envelope in the form of a list of break-point pairs.
- A number that is the factor by which the y values (data segment of the
  break-point pairs) are to be scaled.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :y-min. A number that is the minimum value for all y values after
   scaling. NB The -min/-max arguments are hard-limits only; they do not
   factor into the arithmetic.
 - :y-max. A number that is the maximum value for all y values after
   scaling.
 - :x-scaler. A number that is the factor by which to scale the x-axis
   values of the break-point pairs.
 - :x-min. A number that is the minimum value for all x values after
   scaling. NB: This optional argument can only be used if a value has been
   specified for the :x-scaler.
 - :x-max. A number that is the maximum value for all x values after
   scaling. NB: This optional argument can only be used if a value has been
   specified for the :x-scaler.

RETURN VALUE:

 An envelope in the form of a list of break-point pairs.

EXAMPLE:

```
;;; Scaling only the y values.
(scale-env '(0 53 25 189 50 7 75 200 100 3) 0.5)

=> (0 26.5 25 94.5 50 3.5 75 100.0 100 1.5)

;;; Scaling the y values and setting a min and max for those values
(scale-env '(0 53 25 189 50 7 75 200 100 3) 0.5 :y-min 20 :y-max 100)

=> (0 26.5 25 94.5 50 20 75 100 100 20)

;;; Scaling only the x-axis values
(scale-env '(0 53 25 189 50 7 75 200 100 3) 1.0 :x-scaler 2)

=> (0 53.0 50 189.0 100 7.0 150 200.0 200 3.0)

;;; Scaling the x values and setting a min and max for those values
(scale-env '(0 53 25 189 50 7 75 200 100 3) 1.0 :x-scaler 2 :x-min 9 :x-max 90)

=> (9 53.0 50 189.0 90 7.0 90 200.0 90 3.0)
```

SYNOPSIS:

```
(defun scale-env (env y-scaler &key x-scaler
                                    (x-min most-negative-double-float)
```

```
                                    (y-min most-negative-double-float)
                                    (x-max most-positive-double-float)
                                    (y-max most-positive-double-float))
```

21.71   utilities/secs-to-mins-secs

[utilities] [Functions]

DESCRIPTION:

Convert a number of seconds into a string of the form "24:41.723" where
seconds are always rounded to three decimal places (i.e. milliseconds).

ARGUMENTS:

- the number of seconds

OPTIONAL ARGUMENTS:

keyword arguments:
- :post-mins. The string used to separate minutes and seconds. Default ":"
- :post-secs. The string used to separate seconds and milliseconds.
  Default "."
- :post-msecs. The string used to follow milliseconds. Default ""
- :same-width. Ensure minutes values are always two characters wide, like
  seconds, i.e with a leading 0.
- :round. Round to the nearest second and don't print milliseconds. Default
  NIL.

RETURN VALUE:

A string

EXAMPLE:

```
(secs-to-mins-secs 77.1232145)
"1:17.123"
(secs-to-mins-secs 67.1)
"1:07.100"
(secs-to-mins-secs 67.1 :same-width t)
"01:07.100"
(secs-to-mins-secs 67.1 :same-width t :post-secs "s")
"01:07s100"
(secs-to-mins-secs 67.1 :post-secs "secs" :post-mins "min" :post-msecs "msecs")
"1min07secs100msecs"
```

```
(secs-to-mins-secs 67.7 :same-width t :round t)
"01:08"
```

SYNOPSIS:

```
(defun secs-to-mins-secs (seconds &key
                                   round
                                   (post-mins ":")
                                   (post-secs ".")
                                   (post-msecs "")
                                   (same-width nil))
```

21.72   utilities/semitones

[utilities] [Functions]

DESCRIPTION:

Return the sample-rate conversion factor required for transposing an audio
file by a specific number of semitones. The number of semitones can be
given as a decimal number, and may be positive or negative.

ARGUMENTS:

 - A number of semitones.

OPTIONAL ARGUMENTS:

 - A number that is the factor required to transpose by an octave.
   Default = 2.0.
 - A number that is the number of semitones per octave. Default = 12.

RETURN VALUE:

 A number.

EXAMPLE:

```
;;; Usage with default values
(semitones 3)

=> 1.1892071

;;; Specifying a different number of semitones per octave
```

```
(semitones 3 2.0 13)

=> 1.1734605

;;; Specifying a different factor for transposing by an octave
(semitones 3 4.0)

=> 1.4142135

;;; Fractional semitones are allowed
(semitones 3.72)

=> 1.2397077

;;; Negative semitones are also allowed
(semitones -3.72)

=> 0.80664176
```

SYNOPSIS:

```
(defun semitones (st &optional (octave-size 2.0) (divisions-per-octave 12))
```

21.73   utilities/setf-last

[utilities] [Functions]

DESCRIPTION:

 Change the last element in a given list to a specified new element.

ARGUMENTS:

 - A list.
 - The new last element of that list.

RETURN VALUE:

 Returns the new last element.

EXAMPLE:

```
(let ((l '(1 2 3 4 5)))
  (setf-last l 'dog)
  l)
```

```
=> (1 2 3 4 DOG)
```

SYNOPSIS:

```
(defmacro setf-last (list new-last)
```

21.74   utilities/sort-symbol-list

[utilities] [Functions]

DESCRIPTION:

 Sort a list of symbols alphabetically ascending, case-insensitive.

ARGUMENTS:

 A list of symbols.

RETURN VALUE:

 The same list of symbols sorted alphabetically ascending, case-insensitive.

EXAMPLE:

```
(sort-symbol-list '(Lorem ipsum dolor sit amet consectetur adipiscing))

=> (ADIPISCING AMET CONSECTETUR DOLOR IPSUM LOREM SIT)
```

SYNOPSIS:

```
(defun sort-symbol-list (list)
```

21.75   utilities/splice

[utilities] [Functions]

DESCRIPTION:

 Insert the elements of a first list into a second list beginning at a
 specified index (0-based).

ARGUMENTS:

- A list that contains the elements to be inserted into the second list.
- A list into which the elements of the first argument are to be inserted.
- An integer that is the index within the second list where the elements
  are to be inserted.

RETURN VALUE:

- A list.

EXAMPLE:

(splice '(dog cat goldfish) '(1 2 3 4 5 6 7 8 9) 3)

=> (1 2 3 DOG CAT GOLDFISH 4 5 6 7 8 9)

SYNOPSIS:

(defun splice (elements into-list where)

21.76   utilities/split-groups

[utilities] [Functions]

DESCRIPTION:

Create a list consisting of as many repetitions of a specified number as
will fit into a given greater number, with the last item in the new list
being the value of any remainder.

ARGUMENTS:

- A number that is to be split into repetitions of a specified smaller
  number (the second argument).
- The number that is to be the repeating item in the new list. This number
  must be smaller than the first number.

RETURN VALUE:

A list consisting of repetitions of the specified number, with the last
element being any possible remainder.

EXAMPLE:

(split-groups 101 17)

=> (17 17 17 17 17 16)

SYNOPSIS:

```
(defun split-groups (num divider)
```

21.77   utilities/split-into-sub-groups

[utilities] [Functions]

DESCRIPTION:

Create a new list consisting of sublists made from the elements of the
original flat list, whose lengths are determined by the second argument to
the function.

NB: The lengths given in the second argument are not required to add up to
    the length of the original list. If their sum is less than the original
    list, the resulting list of sublists will only contain a segment of the
    original elements. If their sum is greater than the length of the
    original list, the last sublist in the new list will be shorter than
    the corresponding group value.

ARGUMENTS:

- A flat list.
- A list of integers that are the lengths of the consecutive subgroups
  into which the original list is to be divided.

RETURN VALUE:

A list of lists.

EXAMPLE:

```
;; Used with a list of subgroup lengths whose sum is equal to the length of the
;; original list
(split-into-sub-groups '(1 2 3 4 5 6 7 8 9 10) '(2 2 3 2 1))

=> ((1 2) (3 4) (5 6 7) (8 9) (10))

;; Used with a list of subgroup lengths whose sum is less than the length of the
;; original list
(split-into-sub-groups '(1 2 3 4 5 6 7 8 9 10) '(2 1))

=> ((1 2) (3))
```

```
;; Used with a list of subgroup lengths whose sum is greater than the length of
;; the original list
(split-into-sub-groups '(1 2 3 4 5 6 7 8 9 10) '(2 3 17))

=> ((1 2) (3 4 5) (6 7 8 9 10))
```

SYNOPSIS:

```
(defun split-into-sub-groups (list groups)
```

21.78   utilities/split-into-sub-groups2

[utilities] [Functions]

DESCRIPTION:

```
Create a new list of lists by splitting the original flat list into
sublists of the specified length.

NB: The length given as the second argument is not required to be fit
    evenly into the length of the original flat list. If the original list
    is not evenly divisible by the specified length, the resulting list of
    sublists will contain a final sublist of a different length.
```

ARGUMENTS:

```
- A flat list.
- An integer that is the length of each of the sublists to be created.
```

RETURN VALUE:

```
A list of lists.
```

EXAMPLE:

```
;; The second argument fits evenly into the length of the original list.
(split-into-sub-groups2 '(1 2 3 4 5 6 7 8 9 10 11 12) 3)

=> ((1 2 3) (4 5 6) (7 8 9) (10 11 12))

;; The second argument does not fit evenly into the length of the original
;; list.

(split-into-sub-groups2 '(1 2 3 4 5 6 7 8 9 10 11 12) 5)

=> ((1 2 3 4 5) (6 7 8 9 10) (11 12))
```

SYNOPSIS:

```
(defun split-into-sub-groups2 (list length)
```

21.79   utilities/split-into-sub-groups3

[utilities] [Functions]

DESCRIPTION:

```
Split a given flat list into sublists of the specified length, putting any
remaining elements, if there are any, into the last sublist.
```

ARGUMENTS:

```
- A flat list.
- An integer that is the length of the new sublists.
```

RETURN VALUE:

```
A list of lists.
```

EXAMPLE:

```
(split-into-sub-groups3 '(1 2 3 4 5 6 7 8 9 10 11 12) 3)

=> ((1 2 3) (4 5 6) (7 8 9) (10 11 12))

(split-into-sub-groups3 '(1 2 3 4 5 6 7 8 9 10 11 12) 5)

=> ((1 2 3 4 5) (6 7 8 9 10 11 12))
```

SYNOPSIS:

```
(defun split-into-sub-groups3 (list length)
```

21.80   utilities/split-into-sub-groups4

[utilities] [Functions]

DATE:

```
August 28th 2018, Heidhausen
```

DESCRIPTION:

Split into sub-groups with lengths defined by the integers in the second
argument list. The difference here to related functions is that the given
lengths will repeat circularly until no more elements of the first argument
remain.

ARGUMENTS:

- the list to split into sub-groups
- a list of lengths, to be repeated

RETURN VALUE:

a list of sublists

EXAMPLE:

```
(split-into-sub-groups4 '(1 2 3 4 5 6 7 8 9 10 11 12) '(3 4))
==> ((1 2 3) (4 5 6 7) (8 9 10) (11 12))

(split-into-sub-groups4 '(1 2 3 4 5 6 7 8 9 10 11 12 13 14) '(3 4))
==> ((1 2 3) (4 5 6 7) (8 9 10) (11 12 13 14))

(split-into-sub-groups4 '(1 2 3 4 5 6 7 8 9 10 11 12) '(1 2 3 4))
==> '((1) (2 3) (4 5 6) (7 8 9 10) (11) (12))
```

SYNOPSIS:

```
(defun split-into-sub-groups4 (list lengths)
```

21.81   utilities/srt

[utilities] [Functions]

DESCRIPTION:

Return the semitone transposition for a given sampling rate conversion
factor.

ARGUMENTS:

- A number that is a sample-rate conversion factor.

OPTIONAL ARGUMENTS:

- A number that is the factor required for transposing one octave.
- A number that is the number of scale degrees in an octave.

RETURN VALUE:

 A number.

EXAMPLE:

```
;;; Using the defaults
(srt 1.73)

=> 9.4893

;;; Using a sample-rate conversion factor of 4.0 for the octave and specifying
;;; 13 divisions of the octave
(srt 1.73 4.0 13)

=> 5.14
```

SYNOPSIS:

```
(let ((last8vesize 0)
      (log8ve 0.0)) ;; so we don't have to recalculate each time
  (defun srt (srt &optional (octave-size 2.0) (divisions-per-octave 12)
               ;; MDE Tue Feb  7 16:59:45 2012 -- round so we don't get tiny
               ;; fractions of semitones due to float inaccuracies?
               (round-to 0.0001))
```

21.82   utilities/string-replace

[utilities] [Functions]

DESCRIPTION:

 Replace specified segments of a string with a new specified string.

ARGUMENTS:

- A string that is the string segment to be replaced.
- A string that is the string with which the specified string segment is to
  be replaced.
- The string in which the specified segment is to be sought and replaced.

RETURN VALUE:

 A string.

EXAMPLE:

(string-replace "flat" "\\flat" "bflat clarinet")

=> "b\\flat clarinet"

SYNOPSIS:

(defun string-replace (what with string)

21.83 utilities/swap-elements

[utilities] [Functions]
DESCRIPTION:

 Swap the order of each consecutive pair of elements in a list.

ARGUMENTS:

 - A list.

RETURN VALUE:

 A list.

EXAMPLE:

(swap-elements '(1 2 3 4 5 6 7 8 9 10))

=> (2 1 4 3 6 5 8 7 10 9)

(swap-elements '(1 2 3 4 5 6 7 8 9))

=> (2 1 4 3 6 5 8 7 9)

SYNOPSIS:

(defun swap-elements (list)

21.84 utilities/update-app-src

[utilities] [Functions]

DATE:

 June 1st 2013

DESCRIPTION:

 TEMPORARILY DISABLED DUE TO SVN SERVER ACCESS RESTRICTIONS.
 NB This function currently works in SBCL and CCL on UNIX systems only.

 For users of the slippery chicken app, this function will update the source
 code of the app to the latest in the online subversion (svn) repository.
 An internet connection is therefore necessary.

 The first time it is run it will delete the current source code and
 download all the new source code, so make sure to back up if you've
 modified the source code yourself (not recommended). When it is run from
 then on, it will only update the source code that is out of date.

 Once the source code is updated, you'll need to restart the app or just
 Lisp for the changes to be recompiled.

 NB The first time you call this function, you might get a "certificate
 error". In order to accept the certificate, start the terminal application
 and type the following:

 cd /tmp/
 svn co https://svn.ecdf.ed.ac.uk/repo/user/medward2/sc-tags/sc-latest/src

 That should give you a prompt in the terminal from which you can accept the
 certificate. Then the next time you try it from Lisp the certificate
 should not cause a problem.

 Users without the app can always download the latest source code in a
 terminal by issuing the following command.
 svn co https://svn.ecdf.ed.ac.uk/repo/user/medward2/sc-tags/sc-latest/src

ARGUMENTS:

 The full path to the slippery-chicken application, minus the last slash.
 Remember that this can't include any spaces in file/folder names

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :rm.  The path to the shell 'rm' command.  Default = "/bin/rm"
- :svn.  The path to the shell 'svn' command.  Default = "/usr/bin/svn"
```

RETURN VALUE:

```
The shell return value of the call to SVN, usually 0 on success.
```

EXAMPLE:

```
Running for the first time:
(update-app-src "/tmp/sc-app/slippery-chicken.app")
A    /tmp/sc-app/slippery-chicken.app/Contents/Resources/sc/src/sndfile.lsp
A    /tmp/sc-app/slippery-chicken.app/Contents/Resources/sc/src/osc.lsp
A    /tmp/sc-app/slippery-chicken.app/Contents/Resources/sc/src/osc-sc.lsp
[...]
Checked out revision 3608.
0

or after successfully updating a previously updated version:
...
At revision 3608.
0
```

SYNOPSIS:

```
(defun update-app-src (path-to-app &key (rm "/bin/rm") (svn "/usr/bin/svn"))
```

21.85   utilities/wavelab-to-audacity-marker-file

[utilities] [Functions]

DESCRIPTION:

```
Write a .txt file suitable for import to audacity with the same name and in
the same directory as the file argument.
```

ARGUMENTS:

```
- A string that is the name of a wavelab marker file, including directory
  path and extension.
```

OPTIONAL ARGUMENTS:

- An integer that is the sampling rate of the sound file to which the
  wavelab marker file refers. This value will affect the times of the
  output.

RETURN VALUE:

 Returns T and prints the number of markers read to the listener.

EXAMPLE:

(wavelab-to-audacity-marker-file "/path/to/24-7.mrk" 44100)

=> 51 markers read

SYNOPSIS:

(defun wavelab-to-audacity-marker-file (file &optional (sampling-rate 44100))

21.86   utilities/wrap-list

[utilities] [Functions]

DESCRIPTION:

 Shift the elements of a list to start at a specified position and wrap to
 the beginning of the list to the list's tail.

ARGUMENTS:

 - A list.
 - An integer which is the 0-based position in the original list where the
   new list is to begin.

RETURN VALUE:

 A list.

EXAMPLE:

(wrap-list '(1 2 3 4 5 6 7 8 9) 4)

=> (5 6 7 8 9 1 2 3 4)

SYNOPSIS:

(defun wrap-list (list start)

21.87 utilities/write-list-to-coll

[utilities] [Functions]

DATE:

 Tue 18 Feb 2020 15:38:40 GMT

DESCRIPTION:

 Turn a list of lists into a text file, formatted to be read by the MaxMSP
 [coll] object. This is a bit like gen-max-coll-file (see set-palette.lsp)
 but instead works with any data in a list.

ARGUMENTS:

 - A list of lists in the form '((a b c) (d e f))

OPTIONAL ARGUMENTS:

 keyword arguments
 :file - the output file. Default = "/tmp/sc-max-coll.txt"
 :base - the minimum number for coll indexing. In the resulting output file,
 each list in the list of lists will be preceded by an (increasing) integer
 and a comma. This argument sets the base value of that integer. Default = 0.
 :capitalize - Should any outputted text be capitalized or not?
 Default = NIL.
 :if-exists - what to do if the file already exists. This argument is passed
 to with-open-file. More info here: http://clhs.lisp.se/Body/m_w_open.htm
 Default = :supercede
 :prefix - add a string prefix to the item number. Default = NIL.
 :alt-label - if you do not want the items labels to be consecutive numbers,
 then you can here either provide a list of lists or a function. If a list of
 lists, this must be the same length as the first argument. If a function, it
 must be called with the item number (which increases incrementally from
 base).

RETURN VALUE:

 The output file location

EXAMPLE:

```
(let ((l '((hello!)(how are you?)(very well thank you.)(1 2 3 4))))
     (write-list-to-coll l :base 6))
```

```
=> "/tmp/sc-max-coll.txt"
```

The resulting text file will looks like this when opened:

```
6, hello!;
7, how are you?;
8, very well thank you.;
9, 1 2 3 4;


;; DJR Tue 3 Mar 2020 13:52:34 GMT
(let ((l '((hello!)(how are you?)(very well thank you.)(1 2 3 4))))
      (write-list-to-coll l :base 15
                            :alt-label #'(lambda (count)
                                            (let ((l '(foo bar)))
                                              (nth (mod count 2) l)))
                            :prefix "yes_"))

=> "/tmp/sc-max-coll.txt"
```

The resulting text file will looks like this when opened:

```
yes_bar, hello!;
yes_foo, how are you?;
yes_bar, very well thank you.;
yes_foo, 1 2 3 4;
```

SYNOPSIS:

```
(defun write-list-to-coll (data-list &key (base 0)
                                          (file "/tmp/sc-max-coll.txt")
                                          (capitalize nil)
                                          (if-exists :supersede)
                                          ;; DJR Tue 3 Mar 2020 13:52:34 GMT
                                          (prefix "")
                                          alt-label)
```

22   clm/clm-loops

[Functions]

DESCRIPTION:

 Generate a sound file from an existing specified sound file by shuffling
 and repeating specified segments within the source sound file.

This function was first introduced in the composition "breathing Charlie" (under the name loops): see charlie-loops.lsp in that project for examples.

The first required argument to the function is the name of the sound file, including path and extension, looped. This must be a mono file.

The second required argument (entry-points) is a list of times, in seconds, where attacks (or something significant) happen in the file. These are used to create loop start/end points.

Be careful when doing shuffles as if, e.g., the transpositions list is more than 6 elements, shuffling will take a very long time.

The entry-points are used randomly so that any segment may start at any point and transition to any other segment (i.e. skipping intervening segments, always forwards however). There are always two segments in use at any time. The function randomly selects which segments are used, then a transition (see fibonacci-transitions) from repeated segment 1 to repeated segment 2 is made. Then the next segment is chosen and the process is repeated (i.e. from previous segment 2 to new segment) until the max-start-time (in seconds) is achieved.

fibonacci-transitions are first shuffled and then made into a circular list. Then they are expanded to create the transpositions (each number becomes a series of 1s and 0s--length is the number itself--with a transition from all 0s to all 1s: e.g. (fibonacci-transition 20) -> (0 0 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1) This is then used to select one or the other of the current two segments.

The sample-rate transpositions are simply randomly permutated and selected.

ARGUMENTS:

- The name of a sound file, including path and extension.
- A list of numbers that are time in seconds. These serve as the "entry-points", i.e. loop markers within the file, and delineate the beginning and end of segments that will be shuffled and played back at random in the resulting file.

OPTIONAL ARGUMENTS:

keyword arguments.
- :max-perms. A number that is the maximum number of permutations generated for the transitions. Default = 1000.
- :fibonacci-transitions. A list of numbers that serve as the number of

    steps in each transition from one segment to the next. These numbers will
    be used as the first argument to the call to fibonacci-transition.
    Default = '(34 21 13 8)
- :max-start-time. A number that is the maximum time in second at which a
  segment can start in the resulting sound file. Default = 60.0.
- :output-dir. The directory path for the output file.
  Default = (get-sc-config 'default-dir).
- :srate. The sampling rate. If specified by the user, this will generally
  be a number. By default it takes the CLM global sample-rate, i.e.
  clm::*clm-srate*
- :data-format. The data format of the resulting file. This must be
  preceded by the clm package qualifier. See clm.html for types of data
  formats, such as mus-bshort, mus-l24float etc.
  Default is the whatever the CLM global clm::*clm-data-format* is set to.
- :header-type. The header type of the resulting file. This must be
  preceded by the clm package qualifier. See clm.html for possible header
  types, such as mus-riff, mus-aifc etc. By default it takes the CLM global
  clm::*clm-header-type*.
- :sndfile-extension. A string or NIL. If a string, this will be appended
  to the resulting sound file as a file extension. If NIL, the sound file
  extension will automatically be selected based on the header type. NB:
  This argument does not affect the header type! Default = NIL.
- :channels. An integer that is the number of channels in the resulting
  output. If greater than one, the segments will be automatically panned
  amongst the channels. Default = 1.
- :transpositions. A list of number that are transpositions in
  semitones. These will be shuffled and applied randomly to each
  consecutive segment in the output. Default = '(0).
- :num-shuffles. An integer that will indicate how many times the lists
  passed to fibonacci-transitions and entry-points will be shuffled before
  generating output. Default = - 1.
- :suffix. A string that will be automatically appended to the end of the
  file name. Default = "".
- :src-width. A number that represents the accuracy of the sample-rate
  conversions undertaken for transposition. The higher this number is, the
  more accurate the transposition will be, but the longer it will take to
  process the file. Default = 5.

RETURN VALUE:

 Returns the name of the file generated.

EXAMPLE:

```
;;; A straightforward example with a number of the variables.
(clm-loops "/path/to/sndfile-3.aiff"
```

```
          '(0.180 2.164 4.371 7.575 9.4 10.864)
          :fibonacci-transitions '(1 2 3 4 5)
          :max-perms 7
          :output-dir "/tmp/"
          :channels 1
          :transpositions '(1 12 -12)
          :num-shuffles 3
          :src-width 20)
```

```
=> "/tmp/sndfile-3-loops-from-00m00.180-.wav"
```

SYNOPSIS:

```
#+clm
(defun clm-loops (sndfile entry-points &key
                    (max-perms 1000)
                    (fibonacci-transitions '(34 21 13 8))
                    (max-start-time 60.0)
                    (output-dir (get-sc-config 'default-dir))
                    (srate clm::*clm-srate*)
                    (data-format clm::*clm-data-format*)
                    ;; MDE Fri May 11 15:33:45 2012
                    (header-type clm::*clm-header-type*)
                    ;; MDE Fri May 11 15:34:17 2012 --
                    (sndfile-extension nil)
                    (channels 1)
                    ;; semitones
                    (transpositions '(0))
                    ;; added 31/7/05 to vary the order of
                    ;; entry points, transpositions and
                    ;; fibonacci-transitions (could be 0!)
                    (num-shuffles 1)
                    (suffix "")
                    (src-width 5))
```

23   clm/clm-loops-all

[Functions]

DESCRIPTION:

Similar to clm-loops, but takes a list of lists of entry points (which can
also be generated using the random-loop-points function, for example) and
produces one output sound file for each list of entry points that list
contains.

ARGUMENTS:

- A string that is the name of the source sound file including directory
  path and extension.
- A list of lists of numbers that are entry points (loop markers) in the
  specified source sound file.

OPTIONAL ARGUMENTS:

keyword arguments:
- :max-perms. A number that is the maximum number of permutations generated
  for the transitions. Default = 1000.
- :fibonacci-transitions. A list of numbers that serve as the number of
  steps in each transition from one segment to the next. These numbers will
  be used as the first argument to the call to fibonacci-transition.
  Default = '(34 21 13 8).
- :max-start-time. A number that is the maximum time in seconds at which a
  segment can start in the resulting sound file. Default = 60.0.
- :output-dir. The directory path for the output file.
  Default = (get-sc-config 'default-dir).
- :srate. The sampling rate. If specified by the user, this will generally
  be a number. By default it takes the CLM global sample-rate, i.e.
  clm::*clm-srate*
- :data-format. The data format of the resulting file. This must be
  preceded by the clm package qualifier. See clm.html for types of data
  formats, such as mus-bshort, mus-l24float etc.
  Default is the whatever the CLM global clm::*clm-data-format* is set to.
- :header-type. The header type of the resulting file. This must be
  preceded by the clm package qualifier. See clm.html for possible header
  types, such as mus-riff, mus-aifc etc. By default it takes the CLM global
  clm::*clm-header-type*.
- :sndfile-extension. A string or NIL. If a string, this will be appended
  to the resulting sound file as a file extension. If NIL, the sound file
  extension will automatically be selected based on the header type. NB:
  This argument does not affect the header type! Default = NIL.
- :channels. An integer that is the number of channels in the resulting
  output. If greater than one, the segments will be automatically panned
  amongst the channels. Default = 1.
- :do-shuffles. T or NIL to indicate whether to shuffle the lists passed to
  fibonacci-transitions and entry-points before generating output.
  T = do shuffles. Default = T.
- :start-after. A number. All loops will be excluded that start before this
  number of seconds. Default = -1.0.
- :stop-after. A number. All loops will be excluded that start after this
  number of seconds. Default = 99999999.0.
- :suffix. A string that will be automatically appended to the end of the

```
  file name. Default = "".
- :transpositions. A list of number that are transpositions in
  semitones. These will be shuffled and applied randomly to each
  consecutive segment in the output. Default = '(0).
- :transposition-offset. A number that is an additional number of semitones
  to be added to each transposition value before performing the
  transposition. Default = 0.0.
- :src-width. A number that represents the accuracy of the sample-rate
  conversions undertaken for transposition. The higher this number is, the
  more accurate the transposition will be, but the longer it will take to
  process the file. Default = 5.
```

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(clm-loops-all
 (concatenate 'string
              cl-user::+slippery-chicken-home-dir+
              "test-suite/test-sndfiles-dir-1/test-sndfile-3.aiff")
 '((0.794 0.961 1.061 1.161 1.318 1.436 1.536)
   (0.787 0.887 0.987 1.153 1.310 1.510)
   (0.749 0.889 1.056 1.213 1.413)
   (0.311 0.411 0.611 0.729)
   (0.744 0.884 1.002))
 :max-perms 6
 :fibonacci-transitions '(31 8 21 13)
 :output-dir "/tmp/"
 :channels 1
 :transpositions '(1 12 -12)
 :src-width 20)
```

SYNOPSIS:

```
#+clm
(defun clm-loops-all (sndfile entry-points-list
                      &key
                      (max-perms 1000)
                      (fibonacci-transitions '(34 21 13 8))
                      (max-start-time 60.0)
                      (output-dir (get-sc-config 'default-dir))
                      (srate clm::*clm-srate*)
                      (data-format clm::*clm-data-format*)
```

```
;; MDE Fri May 11 15:33:45 2012
(header-type clm::*clm-header-type*)
;; MDE Fri May 11 15:34:17 2012 --
(sndfile-extension nil)
(channels 1)
(do-shuffles t) ;; see clm-loops
;; exclude all those loops who start before this
;; number of seconds.
(start-after -1.0)
(stop-after 99999999.0)
(suffix "")
;; semitones
;; 6/10/06: using just one list of transpositions passed
;; onto clm-loops created the same tone structure for
;; every file generated (boring).  This list will now be
;; shuffled and 10 versions collected which will then be
;; passed (circularly) one after the other to clm-loops.
(transpositions '(0))
(transposition-offset 0.0)
(src-width 5))
```

24   clm/random-loop-points

[Functions]

DESCRIPTION:

Return a list of lists of randomly generated entry points (loop markers)
for use with clm-loops-all.

This function also produces an output text file containing the same list of
lists. This file is in Lisp syntax and can therefore be accessed using
read-from-file.

ARGUMENTS:

- A string that is the file name, including directory path and extension,
  of the output file to produce.
- A string that is the sound file for which to generate random entry
  points.

OPTIONAL ARGUMENTS:

keyword arguments:

- :min-points. An integer that is the least number of entry points to
  generate for each list. Default = 5.
- :max-points. An integer that is the greatest number of entry points to
  generate for each list. Default = 13.
- :min-dur. A number that is the shortest duration between two entry
  points. Default = 0.05.
- :num-loop-sets. An integer that is the number of lists of entry points to
  generate. Default = 20.
- :scalers. A list of fractions that are durations relative to the min-dur,
  such that, for example, a min-dur of 0.05 with a scaler of 13/8 would
  result in a scaled duration of 0.08125. The fractions in this list will
  be chosen at random when calculating the duration of the next loop
  segment. Default = '(1/1 2/1 3/2 5/3 8/5 13/8).

RETURN VALUE: EXAMPLE:

```
(random-loop-points
 "/tmp/outfile"
 "/path/to/test-sndfile-3.aiff"
 :min-points 3
 :max-points 7
 :min-dur 0.1
 :num-loop-sets 5
 :scalers '(1/1 2/1 3/2 5/3 7/5 11/7 13/11))

=> ((0.789 0.929 1.079) (0.028 0.228 0.368 0.487 0.687) (0.014 0.164 0.321)
    (0.256 0.406 0.524 0.681) (0.069 0.235 0.353 0.472 0.572 0.69))
```

SYNOPSIS:

```
#+clm
(defun random-loop-points (outfile sndfile
                           &key
                           ;; MDE Thu May 17 17:02:15 2012 -- could also be
                           ;; :error or anything else that with-open-file
                           ;; accepts
                           (if-outfile-exists :overwrite)
                           ;; the minimum number of time points for an output
                           ;; loop--number of looped sound segments is 1- this
                           (min-points 5)
                           ;; max number of time points--the actual number of
                           ;; points will be randomly chosen between these two
                           ;; numbers.
                           (max-points 13)
                           ;; minimum duration of a loop segment--this number
                           ;; will actually be used and scaled by scalers
```

```
                              (min-dur 0.05)
                              ;; how many sets of loops should be generated
                              (num-loop-sets 20)
                              ;; scalers for the min-dur: these are all
                              ;; proportions relative to min-dur so if we have
                              ;; 13/8 in this list and min-dur of 0.05 then the
                              ;; duration for such a segment would be 0.08125.
                              ;; these will be chosen at random when calculating
                              ;; the next loop segment duration
                              (scalers '(1/1 2/1 3/2 5/3 8/5 13/8)))
```

25   get-nearest-note/player

[Methods]

DATE:

 Tue 11 Aug 2020 17:16:10 BST

DESCRIPTION:

 Like get-note, but will return the nearest note before or after the
 specified bar / event number place from, potentially, any bar.

ARGUMENTS:

 - an sc object
 - the bar number to start looking
 - the event number within the bar to start looking

OPTIONAL ARGUMENTS:

 None.

RETURN VALUE:

 If found, the nearest sounding event in the stated player's part and the
 number of events away (positive for forwards, negative for backwards),
 otherwise NIL.

EXAMPLE:

```
(let ((mini
```

```
        (make-slippery-chicken
         '+mini+
         :ensemble '(((sax (alto-sax :midi-channel 1))
                      (vn (violin :midi-channel 1))))
         :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
         :set-map '((1 (1 1 1 1 1))
                    (2 (1 1 1 1 1))
                    (3 (1 1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5))))
                             (2 ((((4 4) (w)))))
                             (3 ((((4 4) (h) q e (s) s))
                                 :pitch-seq-palette ((1 2 5)))))
         :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                             (vn (2 2 2 2 2))))
                         (2 ((sax (2 2 2 2 2))
                             (vn (2 2 2 2 2))))
                         (3 ((sax (3 3 3 3 3))
                             (vn (2 2 2 2 2)))))))))
    (get-nearest-note mini 11 1 'sax))
=>
EVENT: start-time: 42.000, end-time: 43.000,
       duration-in-tempo: 1.000,
       compound-duration-in-tempo: 1.000,
       amplitude: 0.700
       bar-num: 11, marks-before: NIL,
       tempo-change: NIL
       instrument-change: NIL
       display-tempo: NIL, start-time-qtrs: 42.000,
       midi-time-sig: NIL, midi-program-changes: NIL,
       midi-control-changes: NIL,
       8va: 0, player: SAX
       asco-label: NIL, asco-msgs: NIL
       set-ref: (1)
       pitch-or-chord:
PITCH: frequency: 164.814, midi-note: 52, midi-channel: 1
       pitch-bend: 0.0
       degree: 104, data-consistent: T, white-note: E3
       nearest-chromatic: E3
       src: 0.62996054, src-ref-pitch: C4, score-note: E3
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 3, c5ths: 0, no-8ve: E, no-8ve-no-acc: E
       show-accidental: T, white-degree: 30,
       accidental: N,
```

```
      accidental-in-parentheses: NIL, marks: NIL,
      marks-before: NIL, amplitude: NIL
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: E3, tag: NIL,
data: E3
**************

      written-pitch-or-chord:
PITCH: frequency: 277.183, midi-note: 61, midi-channel: 1
      pitch-bend: 0.0
      degree: 122, data-consistent: T, white-note: C4
      nearest-chromatic: CS4
      src: 1.0594631, src-ref-pitch: C4, score-note: CS4
      qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
      micro-tone: NIL,
      sharp: T, flat: NIL, natural: NIL,
      octave: 4, c5ths: 2, no-8ve: CS, no-8ve-no-acc: C
      show-accidental: T, white-degree: 35,
      accidental: S,
      accidental-in-parentheses: NIL, marks: NIL,
      marks-before: NIL, amplitude: NIL
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: CS4, tag: NIL,
data: CS4
**************
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
        is-whole-bar-rest: NIL,
        score-rthm: 4.0, undotted-value: 4, num-flags: 0, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 4, tuplet-scaler: 1, bar-pos: 1,
        grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q
**************

1
```

SYNOPSIS:

```
(defmethod get-nearest-note ((sc slippery-chicken) bar-num event-num player)
```

26   globals/+slippery-chicken-config-data+

[Global Parameters]

DESCRIPTION:

```
 A global to hold various user-settable configuration settings.  Use e.g.
 (set-sc-config 'default-dir "~/Desktop") or (get-sc-config 'default-dir) to
 set or query the settings.
```

SYNOPSIS:

```
(defparameter +slippery-chicken-config-data+
  (make-instance
   'assoc-list
   :id 'slippery-chicken-config-data
   :data
   ;; MDE Sun Mar 25 10:39:07 2012 -- The following two are used in
   ;; pitch-seq::get-notes to indicate which lowest number in a pitch-seq would
   ;; indicate that we should select the highest or lowest notes possible for
   ;; the instrument/set.
   ;;
   ;; The first is used to indicate the lowest number in a pitch-seq that would
   ;; indicate that the get-notes algorithm should select the highest notes
   ;; possible for the instrument/set.
   '((pitch-seq-lowest-equals-prefers-high 5)
     ;; the lowest number in a pitch-seq that would indicate that the get-notes
     ;; algorithm should select the lowest notes possible for the
     ;; instrument/set.
     (pitch-seq-lowest-equals-prefers-low 1)
     ;; MDE Wed Oct 10 16:35:10 2018 -- until October 2018 an error was
     ;; signalled if the were no pitches in a set for an instrument which should
     ;; be playing. We can now have a rest-sequence generated instead
     (pitch-seq-no-pitches-error t)
     ;; MDE Thu Oct  8 16:27:19 2020, Heidhausen -- issue a warning if we scale
     ;; a rhythm/event to longer than 10xwhole?
     (rhythm-scale-warning t)
     ;; MDE Sat Aug  3 16:39:40 2019 -- should shorten-large-fast-leaps issue a
     ;; warning or not?
     (shorten-large-fast-leaps-warning t)
```

```
;; MDE Tue Jul 16 11:17:23 2019 -- if NIL don't even warn about no pitches
;; being available
(pitch-seq-no-pitches-warning t)
;; Whether to automatically open EPS files generated with CMN via
;; cmn-display.  Currently only works with SBCL and CCL on Mac OSX.
(cmn-display-auto-open #+sc-auto-open T #-sc-auto-open nil)
;; Whether to automatically open PDF files generated with via lp-display.
;; Currently only works with SBCL and CCL on Mac OSX.
(lp-display-auto-open #+sc-auto-open T #-sc-auto-open nil)
;; Whether to automatically open MIDI files generated with via midi-play.
;; Currently only works with SBCL and CCL on Mac OSX.
(midi-play-auto-open #+sc-auto-open T #-sc-auto-open nil)
;; MDE Sat Dec  7 11:56:24 2019 -- if a pitch has a mark included in this
;; list, it will be ignored
(midi-play-ignore-marks nil)
;; The default directory for output of sound files, EPS files, and
;; Lilypond files. Don't forget the trailing slash (i.e. "/tmp/" not
;; "/tmp").  Bear in mind that on OSX the /tmp directory is emptied upon
;; reboot so you shouldn't store any files you'd like to keep in there.
(default-dir "/tmp/")
;; The full path to the lilypond command.  We need to set this if we'll
;; call lp-display, i.e. if we want to automatically call Lilypond and
;; open the resultant PDF directly from Lisp.  The default should work if
;; you have the Lilypond app in your Applications folder on OSX.
(lilypond-command
 "/Applications/LilyPond.app/Contents/Resources/bin/lilypond")
;; The default amplitude for all events that don't have amplitude/dynamic
;; set via some means such as marks.
(default-amplitude 0.7)
;; in init-instance of sc class, warn if we can't call sc-init (because it
;; will be called explicitly later)
(warn-no-sc-init t)
(warn-unused-instruments t)
;; whether to warn when there's no CMN mark for a given Lilypond mark
(warn-no-cmn-mark t)
;; sim for Lilypond
(warn-no-lp-mark t)
(warn-no-xml-mark t)
;; warn if a pitch goes above midi note 119?
(warn-high-pitch t)
;; warn if a pitch goes below octave number -1
(warn-low-pitch t)
(warn-fingering t)
;; Bar number offsets for CMN
(cmn-bar-num-dx-for-sc -0.2)
(cmn-bar-num-dy-for-sc 1.2)
```

```
;; MDE Wed Feb  8 16:50:31 2017 -- string-chord-selection-fun (used by
;; default for selecting chords for violin, viola, cello) will call
;; best-string-diad. This tries to find a diad within a certain
;; range. These are the min/max
(best-string-diad-range (7 11))
(best-string-diad-microtones nil)
;; MDE Sat May 10 12:47:25 2014 -- whether to issue warning when we set
;; the asco-msgs slot of a rest event (because they will only be written
;; to an antescofo~ file if this happens to be a rest in the part we're
;; following and we can't know this in advance).
(asco-msg-rest-warning t)
;; if we've added, say, an antescofo~ label to an event with a rehearal
;; letter, we'll get a warning as we can only have one antescofo label per
;; NOTE (though it's not an error to have two, the 2nd will be ignored).
(asco-two-labels-warning t)
;; the default spectral data used in the chord class for things such as
;; dissonance and spectral centroid calcuation. See spectra.lsp for more
;; details about references into +slippery-chicken-spectra+ (such as
;; akoustik-piano-spectra) or see the get-spectrum function in chord.lsp
;; for a description of the format this data can take. Currently
;; slippery-chicken defines three spectra: akoustik-piano-spectra,
;; clm-piano-spectra, and violin-ensemble-spectra. Bear in mind that if
;; you change this it would be best to start slippery-chicken again (or at
;; least regenerated a piece's data) so that chord's/set's
;; dissonance/centroid values are recalculated with the new spectra.
(default-spectra akoustik-piano-spectra)
;; MDE Tue Apr 19 12:05:49 2016 -- for debugging purposes, print info
;; about the selection of pitches for an instrument given a pitch-seq and
;; set etc. (i.e. data passed to pitch-seq's get-notes method). Note that
;; chords will be printed as a sublist (i.e. in parentheses).
(verbose-pitch-selection nil)
;; MDE Thu Oct 18 11:46:46 2018 -- since Sept. 18 the slippery-chicken
;; method update-slots also calls handle-ties. This will break some older
;; pieces so allow that to be skipped if necessary.
(update-slots-handles-ties t)
;; font size for CMN bar numbers
(cmn-bar-num-size-for-sc 6))))
```

27   osc-sc/osc-call

[Functions]

DESCRIPTION:

Allow OSC (over UDP) messages to be sent for processing. The function

waits for input and processes in an endless loop; send 'quit' to stop the
function and return to the interpreter. Messages the function doesn't
understand will be ignored after a warning being printed.

As this function only terminates when a 'quit' message is sent via OSC, the
only way to quit from within Lisp is to send the Interrupt Command (usually
Control-C, twice). In that case, the open sockets will remain open, and
only closed before reopening the next time this function is called.

Lisp code can be sent, e.g. in MaxMSP via a message, including
opening/closing parentheses and nested calls; symbols should be quoted as
per usual.

The first two tokens in the list must be /osc-sc and a (usually unique)
identifer e.g. ((/osc-sc 1060-osc-sc-eval (print 'dog))) Both of these
tokens will be sent back over OSC in a list, along with the result of the
Lisp call. If you're using MaxMSP I would recommend using
osc-sc-eval.maxpat (see below) to evaluate your Lisp code as this will
package it up with the right tokens and send the evaluated result out of
its outlet without causing conflicts with other instances of itself.

For an example MaxMSP patch, see osc-test.maxpat in the examples folder of
the documentation: http://michael-edwards.org/sc/examples/osc-test.maxpat
You'll also need http://michael-edwards.org/sc/examples/osc-sc-eval.maxpat

NB: Currently only works in SBCL.
    Some lists (e.g. those including strings/symbols) might not be
    recognised as lists by MaxMPS's [route], so process them directly after
    [fromsymbol].

OPTIONAL ARGUMENTS:

keyword arguments:
- :listen-port. The UDP port to listen to for messages. Default = 8090.
- :send-ip. The IP address to send UDP messages back out on.
   Default = #(127 0 0 1))
- :send-port. The UDP port to send messages back out on. Default = 8091.
- :print. Print messages as they arrive. Default = NIL.

RETURN VALUE:

T

SYNOPSIS:

```
(defun osc-call (&key
```

```
                  (listen-port 8090)
                  (send-ip #(127 0 0 1))
                  (print nil)
                  (send-port 8091))
```

28   sc/named-object

[Classes]

NAME:

 named-object

 File: named-object.lsp

 Class Hierarchy: None: base class of all slippery-chicken classes.

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the named-object class which is the
                    base class for all of the slippery-chicken classes.

                    The data slot of the named-object class and its
                    subclasses generally holds the original data passed when
                    creating the object. In anything but the simplest of
                    classes this may quickly become out-of-date as the object
                    is manipulated, but is nevertheless retained so that a)
                    the user can see what data was used to create an object,
                    and b) the user can derive new objects from an object's
                    original data. Data relevant to a specific subclass is
                    often stored in slots other than :data, e.g. bars,
                    rhythms, etc. so the user should not be alarmed if the
                    data slot itself does not seem to reflect changes made to
                    an object.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 4th December 2000

 $$ Last modified: 17:20:12 Fri Sep 25 2020 CEST

 SVN ID: Id
```

## 28.1   named-object/activity-levels

*[ named-object ] [ Classes ]*

**NAME:**

```
 activity-levels

 File:              activity-levels.lsp

 Class Hierarchy:   named-object -> activity-levels

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           Class used in rthm-chain.  Used on a call-by-call basis
                    to determine (deterministically) whether a process is
                    active or not (boolean).

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     4th February 2010

 $$ Last modified:  19:28:24 Thu Sep 24 2020 CEST

 SVN ID: $Id$
```

28.1.1   activity-levels/active

[activity-levels] [Methods]

DESCRIPTION:

Returns t or nil depending on whether we're active at this point. The
object remembers where we were last time; this means if we change level
before getting to the end of a ten-list, we'll pick up where we left off
next time we return to that level. <level> can be a floating point number:
in this case it will be rounded. But <level> must be between 0 and 10,
where 0 is always inactive, 10 is always active, and anything inbetween
will use the data lists circularly.

ARGUMENTS:

- the activity-levels object

OPTIONAL ARGUMENTS:

- the activity-level number we want to test. Although optional, it's
  expected that this argument will usually be defined. Between 0 and 10.
  Default = 5.

RETURN VALUE:

 T or NIL

EXAMPLE:

```
(let ((al (make-al)))
  (print (loop for i below 15 collect (active al 0)))
  (print (loop for i below 15 collect (active al 5)))
  (print (loop for i below 15 collect (active al 1)))
  (print (loop for i below 15 collect (active al 9)))
  (loop for i below 15 collect (active al 10)))

=>
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
(T T NIL NIL T NIL T T NIL NIL NIL T NIL T NIL)
(T NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL T NIL)
(T T NIL T T T T T T T T T T T T NIL)
(T T T T T T T T T T T T T T T)
```

SYNOPSIS:

```
(defmethod active ((al activity-levels) &optional level)
```

28.1.2   activity-levels/make-al

[activity-levels] [Functions]

DESCRIPTION:

 Make an activities-level object for determining (deterministically) on a
 call-by-call basis whether a process is active or not (boolean). This is
 determined by nine 10-element lists (actually three versions of each) of
 hand-coded 1s and 0s, each list representing an 'activity-level' (how
 active the process should be). The first three 10-element lists have only
 one 1 in them, the rest being zeros. The second three have two 1s,
 etc. Activity-levels of 0 and 10 would return never active and always
 active respectively.

ARGUMENTS:

 None required.

OPTIONAL ARGUMENTS:

 start-at (default NIL): which of the three 10-element lists to start with
 (reset to). Should be 0, 1, or 2 though if NIL will default to 0.

RETURN VALUE:

 The activities-level object.

SYNOPSIS:

(defun make-al (&optional start-at id)

28.1.3   activity-levels/reset

[activity-levels] [Methods]

DESCRIPTION:

 Reset the activity-levels object to restart at the first element of the 1st
 (or user-specified) 10-element list.

ARGUMENTS:

 The activity-levels object.

OPTIONAL ARGUMENTS:

 start-at: should be between 0 and 2; it indicates which of the 10-lists
 we're going to start with. Default = 0.

RETURN VALUE:

 T

SYNOPSIS:

(defmethod reset ((al activity-levels) &optional (start-at 0) ignore)

28.2   named-object/activity-levels-env

[named-object] [Classes]

NAME:

```
 activity-levels

 File:              activity-levels-env.lsp

 Class Hierarchy:   named-object -> activity-levels -> activity-levels-env

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           An extension of the activity-levels class to incorporate
                    a restarting envelope which is used to determine the
                    current 'level' argument for a call to the active method.

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     30th June 2015

 $$ Last modified:  19:28:37 Thu Sep 24 2020 CEST

 SVN ID: $Id: activity-levels.lsp 5048 2014-10-20 17:10:38Z medward2 $
```

28.2.1   activity-levels-env/active

[activity-levels-env] [Methods]

DESCRIPTION:

```
 Similar to the parent class method except that the level argument in that
 class is here supplied by the envelope and cycle-length.
```

ARGUMENTS:

```
 - the activity-levels-env object
```

OPTIONAL ARGUMENTS:

```
 - T or NIL to print debugging information.
```

RETURN VALUE:

 T or NIL to indicate whether we're 'active' or not.

EXAMPLE:

```
;;; we can see here that with a cycle-length of 10, calling active 31 times
;;; means we go through the envelope three times before arriving back at 0.
(let ((ale (make-ale '(0 0 2 10 3 1) 10 2)))
  (loop repeat 31 collect (active ale t)))
->
current: 0 level: 0.0
current: 1 level: 1.6666667
current: 2 level: 3.3333335
current: 3 level: 5.0
current: 4 level: 6.666667
current: 5 level: 8.333333
current: 6 level: 10.0
current: 7 level: 7.0
current: 8 level: 4.0
current: 9 level: 1.0
current: 0 level: 0.0
current: 1 level: 1.6666667
current: 2 level: 3.3333335
current: 3 level: 5.0
current: 4 level: 6.666667
current: 5 level: 8.333333
current: 6 level: 10.0
current: 7 level: 7.0
current: 8 level: 4.0
current: 9 level: 1.0
current: 0 level: 0.0
current: 1 level: 1.6666667
current: 2 level: 3.3333335
current: 3 level: 5.0
current: 4 level: 6.666667
current: 5 level: 8.333333
current: 6 level: 10.0
current: 7 level: 7.0
current: 8 level: 4.0
current: 9 level: 1.0
current: 0 level: 0.0
(NIL NIL NIL NIL T T T T NIL NIL NIL NIL NIL T NIL T T T T NIL NIL NIL NIL NIL
 NIL T T T NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod active :around ((ale activity-levels-env) &optional print)
```

28.2.2   activity-levels-env/make-ale

[activity-levels-env] [Functions]

DESCRIPTION:

```
For an overview of what this might be used for and how it's done in its
basic form, see the description to make-al in the parent class
'activity-levels'
```

ARGUMENTS:

```
None required.
```

OPTIONAL ARGUMENTS:

```
- the envelope in x y form. y values should range from 0 to 10, as with the
  level argument to an 'active' call in the parent class. x range can be
  any arbitrary value but start with 0 to get full-range results.
- the cycle length: the envelope repeats after a specified cycle length, so
  that no matter how many times you call 'active' you always get results.
- start-at (default NIL): which of the three 10-element lists in the parent
  class to start with (reset to).  Should be 0, 1, or 2, though if NIL will
  default to 0.
```

RETURN VALUE:

```
The activities-level-env object.
```

SYNOPSIS:

```
(defun make-ale (&optional (env '(0 5 100 5)) (cycle-length 101) start-at)
```

28.3   named-object/control-wave

[named-object] [Classes]

NAME:

```
control-wave

 File:          control-wave.lsp

 Class Hierarchy:  named-object -> control-wave
```

```
Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          The creation of waveforms such as sine, cosine, sawtooth,
                  triangle, square, and pulse train, for the control of
                  musical parameters. See
                  http://michael-edwards.org/wp/?p=977 for context and
                  example code.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    July 6th 2016, Essen Werden, Germany

$$ Last modified:  08:40:56 Thu Mar 26 2020 CET
```

28.4   named-object/control-wave-ins

[named-object] [Classes]

NAME:

```
control-wave

File:             control-wave-ins.lsp

Class Hierarchy:  no classes defined

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          The creation of waveforms such as sine, cosine, sawtooth,
                  triangle, square, and pulse train, for the control of
                  musical parameters. See
                  http://michael-edwards.org/wp/?p=977 for context and
                  example code.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    March 26th 2020, Heidhausen, Germany
                  (taken out of control-wav.lsp)

$$ Last modified:  09:12:54 Thu Mar 26 2020 CET
```

28.5   named-object/linked-named-object

[named-object] [Classes]

NAME:

```
linked-named-object

File:              linked-named-object.lsp

Class Hierarchy:   named-object -> linked-named-object

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Extension of named-object class to provide slots for the
                   previous and next objects in a recursive-assoc-list.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     March 10th 2002

$$ Last modified:  17:47:47 Wed Jan 31 2018 CET

SVN ID: $Id$
```

28.5.1   linked-named-object/bar-holder

[linked-named-object] [Classes]

NAME:

```
bar-holder

File:              bar-holder.lsp

Class Hierarchy:   named-object -> linked-named-object -> bar-holder

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           This class is meant to be sub-classed by piece, section
                   and sequence, all of which hold each other or, ultimately
                   a list of bars with relevant rhythms, timings, pitches
```

```
                        etc.

 Author:              Michael Edwards: m@michael-edwards.org

 Creation date:       16th February 2002

 $$ Last modified:    16:08:50 Tue Jun  9 2020 CEST

 SVN ID: $Id$
```

28.5.2   bar-holder/change-pitches

[bar-holder] [Methods]

DESCRIPTION:

Change the pitches in the specified bars to the specified new pitches.

NB: This method requires that full bars be given, even if not all pitches
    are being changed.

ARGUMENTS:

- A bar-holder object (such as the PIECE slot within a slippery-chicken
  object).
- The ID of the player whose part is to be changed.
- An integer that is the number of the first bar in which pitches are to be
  changed.
- A list of lists of note-name symbols, each sublist representing a
  consecutive bar and containing the same number of note-name symbols as
  there are rhythms in that bar. A NIL in these lists means no change is to
  be made to the corresponding rhythm or bar (see example below). NB: This
  method counts tied notes rather than just attacked notes.

OPTIONAL ARGUMENTS:

 keyword arguments:
- :use-last-octave. T or NIL to indicate whether the method is to require
  that each note-name symbols in the <new-pitches> list has an octave
  indicator. If this argument is set to NIL, each note-name symbol must
  have an octave indicator (e.g., the 4 in c4). If this argument is set to
  T, only the first note-name symbol in the bar is required to have an
  octave indicator, and all subsequent note-name symbols without octave
  indicators will use the last octave indicated; e.g. '((a3 b g cs4)). NB:
  This feature does not work with chords. Default = T.

- :written. T or NIL to indicate whether these are the written or sounding
  notes for a transposing instrument. Default = NIL.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;;; NIL indicates that no change is to be made; this applies to single rhythms
;;; as well as entire bars.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((c2 d2 e2 f2 g2 a2 b2
                            c3 d3 e3 f3 g3 a3 b3
                            c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e (s) s))
                                :pitch-seq-palette ((1 (2) 3 4)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (change-pitches (piece mini) 'cl 2 '((c4 d4 e4 f4)))
  (change-pitches (piece mini) 'vc 3 '((c3 d e f) nil (g3 nil b c4))))

=> T
```

SYNOPSIS:

```
(defmethod change-pitches ((bh bar-holder) player start-bar new-pitches
                           &key (use-last-octave t) written)
```


28.5.3   bar-holder/delete-all-marks

[bar-holder] [Methods]

DESCRIPTION:

 Delete all marks from the MARKS slots of all events in the specified
 measure range of a given bar-holder object and set the slot to NIL.

 This method always applies to full bars.

ARGUMENTS:

- A bar-holder object.
- An integer that is the number of the first bar from which all marks are
  to be deleted.
- An integer that is the number of consecutive bars including the first
  bar from which all marks are to be deleted.
- The ID of the player from whose part the marks are to be deleted.

RETURN VALUE:

Always returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((c2 d2 e2 f2 g2 a2 b2
                            c3 d3 e3 f3 g3 a3 b3
                            c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e (s) s))
                               :pitch-seq-palette ((1 (2) 3 4))
                               :marks (a 1 s 2 te 3 as 4))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (delete-all-marks (piece mini) 2 2 'vc))
```

=> T

SYNOPSIS:

```
(defmethod delete-all-marks ((bh bar-holder) start-bar num-bars player)
```

28.5.4   bar-holder/get-note

[bar-holder] [Methods]

DESCRIPTION:

Return the event object (or pitch object, if accessing a note within a
chord) from a specified bar and note within a given bar-holder object.

ARGUMENTS:

- A bar-holder object (e.g. PIECE slot of a slippery-chicken object).
- An integer that is the 1-based number of the bar from which the note is
  to be retrieved.
- An integer or two-item list of integers that is the 1-based number of the
  note to retrieve within the specified bar. If an integer, the entire
  event object is retrieved. A two-item list of integers is used to
  retrieve a specific note from within a chord, in the form '(2 1), where 2
  is the second note (or non-rhythm event) in the bar, and 1 is the first
  note in the chord counting from the bottom. NB: This argument also counts
  tied notes, not just attacked notes.
- The ID of the player from whose part the note is to be retrieved.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether, when accessing a pitch in a chord, to
  return the written or sounding pitch. T = written. Default = NIL.

RETURN VALUE:

An event object, or single pitch object if accessing a note within a
chord.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((c2 d2 e2 f2 g2 a2 b2
                            c3 d3 e3 f3 g3 a3 b3
                            c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 (2) 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (print (get-note (piece mini) 3 '(2 1) 'vc)) ; single pitch within a chord
  (print (get-note (piece mini) 3 2 'vc)) ; entire chord event
  (print (get-note (piece mini) 5 3 'cl)))
```

SYNOPSIS:

```
(defmethod get-note ((bh bar-holder) bar-num note-num player &optional written)
```

28.5.5   bar-holder/piece

[bar-holder] [Classes]

NAME:

```
 piece

 File:              piece.lsp

 Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                    circular-sclist -> assoc-list -> recursive-assoc-list ->
                    piece
                                  AND

                    named-object -> linked-named-object -> bar-holder ->
                    piece

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           Implementation of the piece class which holds all the
                    note information for a whole piece in the form of
                    sections (possibly subsections), which then contain
                    player-sections, sequenzes and rthm-seq-bars.

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     16th February 2002

 $$ Last modified:  14:31:11 Tue Aug  4 2020 CEST

 SVN ID: $Id$
```

28.5.6   piece/delete-sequenzes

[piece] [Methods]

DESCRIPTION:

```
 Delete one or more consecutive sequenz objects from a given piece object by
 specifying any bar number within the first sequenz object to be deleted.

 This method deletes the whole sequenz object which contains the bar with a
 given number.
```

NB: This method only deletes the sequenz object for the specified player,
    so the remaining players will have a different structure, making MIDI
    or printable output impossible. The user must be sure that each section
    has the same number of sequenzes of identical time-signature structure
    in each section.

NB: The user must call the update-slots method to ensure that changes to
    the NUM-BARS slot etc. are reflected in the given slippery-chicken
    object.

ARGUMENTS:

 - A piece object.
 - An integer that is the number of the bar for which the containing sequenz
   is to be deleted.
 - The ID of the player from whose part the sequenz is to be deleted.

OPTIONAL ARGUMENTS:

 - An integer that is the number of consecutive sequenz objects to delete,
   including the first sequenz indicated by the <bar-num> argument.
   Default = 1.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; Print the number of sequenz objects contained in section 2 of each player's
;;; part, delete two sequenz objects from each part in that section, and print
;;; the number of sequenz objects again to see the difference. Update the slots
;;; and call cmn-display for printable output.
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
```

```
                              (2 ((((4 4) h h))
                                   :pitch-seq-palette ((1 2)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((hn (2 2 2 2 2))
                            (vc (2 2 2 2 2))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (print (length (get-data-data 'hn (get-section mini 2))))
  (print (length (get-data-data 'vc (get-section mini 2))))
  (delete-sequenzes (piece mini) 8 'hn 2)
  (delete-sequenzes (piece mini) 8 'vc 2)
  (print (length (get-data-data 'hn (get-section mini 2))))
  (print (length (get-data-data 'vc (get-section mini 2))))
  (update-slots mini)
  (cmn-display mini))
```

SYNOPSIS:

```
(defmethod delete-sequenzes ((p piece) bar-num player &optional (how-many 1))
```

28.5.7   piece/get-nth-sequenz

[piece] [Methods]

DESCRIPTION:

Get the sequenz object from a specified section of a piece object by
specifying a position index and a player.

NB: This is the primary method for accessing player sequences, as it
    handles cases in which a player doesn't play in a sequence, and it is
    called automatically by slippery-chicken.

    When the specified player has no note events in the specified sequence
    and the optional argument <create-rest-seq> is set to T, this method
    creates a rest sequence (one that consists of the correct number of
    bars with the right time signatures, but in which the bars are only
    rest bars) based on a simultaneous sequence in one of the playing
    instruments.

ARGUMENTS:

- A piece object.
- The ID of the section in from which the sequenz object is to be

     returned.
 - The ID of the player from whose part the sequenz object is to be
   returned.
 - An integer that is the index (position) of the desired sequenz object
   within the given section. This number is 0-based.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to convert sequenz objects that are NIL
   (i.e., the specified player has no events in the specified sequenz) to
   sequenz objects consisting of full-bar rests. T = create rest sequences.
   Default = T. NB: This argument is already called by slippery-chicken with
   a value of T, so has no effect when used as a post-generation editing
   method and can be thus considered for internal use only.

RETURN VALUE:

 Returns a sequenz object.

EXAMPLE:

```
;;; Returns a sequenz object
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((hn (nil nil nil nil nil))
                            (vc (1 1 1 1 1))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (get-nth-sequenz (piece mini) 3 'hn 2))

=>

SEQUENZ: pitch-curve: (1 2 3 4 5)
RTHM-SEQ: num-bars: 1
```

```
        num-rhythms: 5
        num-notes: 5
        num-score-notes: 5
        num-rests: 0
        duration: 4.0
        psp-inversions: NIL
        marks: NIL
        time-sigs-tag: NIL
        handled-first-note-tie: NIL
      (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: (1), next: NIL
BAR-HOLDER:
        start-bar: 13
        end-bar: 13
        num-bars: 1
        start-time: 48.0
        end-time: 52.0
        start-time-qtrs: 48.0
        end-time-qtrs: 52.0
        num-notes (attacked notes, not tied): 5
        num-score-notes (tied notes counted separately): 5
        num-rests: 0
        duration-qtrs: 4.0
        duration: 4.0 (4.000)
```

SYNOPSIS:

```
(defmethod get-nth-sequenz ((p piece) section player seq-num ; 0-based
                            &optional (create-rest-seq t)))
```

28.5.8   piece/get-sequenz-from-bar-num

[piece] [Methods]

DESCRIPTION:

Get the specified sequenz object located at a specified bar-number location
of a specified player's part in a given piece object.

ARGUMENTS:

 - A piece object.
 - An integer that is the number of the bar from which to return the sequenz
   object.

- The ID of the player from whose part the sequenz object is to be
  returned.

RETURN VALUE:

A sequenz object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (get-sequenz-from-bar-num (piece mini) 7 'vc))

                 =>
                 SEQUENZ: pitch-curve: (1 2 3 4 5)
                 RTHM-SEQ: num-bars: 1
                 num-rhythms: 5
                 num-notes: 5
                 num-score-notes: 5
                 num-rests: 0
                 duration: 4.0
                 psp-inversions: NIL
                 marks: NIL
                 time-sigs-tag: NIL
                 handled-first-note-tie: NIL
                 (for brevity's sake, slots pitch-seq-palette and bars are not printed)
                 SCLIST: sclist-length: 3, bounds-alert: T, copy: T
                 LINKED-NAMED-OBJECT: previous: NIL, this: (1), next: NIL
                 BAR-HOLDER:
                 start-bar: 7
                 end-bar: 7
```

```
                    num-bars: 1
                    start-time: 24.0
                    end-time: 28.0
                    start-time-qtrs: 24.0
                    end-time-qtrs: 28.0
                    num-notes (attacked notes, not tied): 5
                    num-score-notes (tied notes counted separately): 5
                    num-rests: 0
                    duration-qtrs: 4.0
                    duration: 4.0 (4.000)
```

SYNOPSIS:

```
(defmethod get-sequenz-from-bar-num ((p piece) bar-num player)
```

28.5.9 piece/insert-bar

[piece] [Methods]

DESCRIPTION:

 Insert a rthm-seq-bar object into an existing piece object.

 NB: As this is a post-generation editing method when used with this class,
     the rthm-seq-bar object must consist of event objects (with pitches),
     not just rhythm objects. If used to insert a bar into an isolated
     rthm-seq object (not a sequenz object), the inserted rthm-seq-bar could
     then consist of rhythm objects rather than events.

 NB: Slippery chicken does not check to ensure that a new bar is inserted
     for each player at a given point; this is up to the user.

 NB: The user must call the update-slots method to ensure that changes to
     the num-bars slot etc. are reflected in the given slippery-chicken
     object.

ARGUMENTS:

 - A piece object.
 - A rthm-seq-bar object.
 - An integer that is the bar-number within the rthm-seq-bar object before
   which the new bar is to be inserted.
 - NB: The optional arguments are actually required for use with this class.

OPTIONAL ARGUMENTS:

NB: The optional arguments are actually required for use with this class.
- An integer that is the section of in which the bar is to be inserted.
- The ID of the player into whose part the new bar is to be added.
- An integer that is the number of the sequenz object into which the bar is
  to be inserted. This is one-based.
- A list of integers that are the curve for a pitch-seq object to be
  applied to the bar that is to be inserted.

RETURN VALUE:

 T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (2 ((((4 4) h h))
                                :pitch-seq-palette ((1 2)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((hn (1 1 1 1 1))
                            (vc (1 2 1 1 1))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))
      (new-bar (make-rthm-seq-bar '((4 4) (w)))))

  (fill-with-rhythms new-bar (loop for r in '(h q. e)
                                   for p in '(c4 e4 g4)
                                   collect (make-event p r)))
  ;; slippery-chicken object has 15 bars  (print (num-bars mini))
  ;; print the number of bars in the sequenz in piece=mini, section=2,
  ;; seq=2 (0=based), player='hn.
  ;; has 1 bar
  (print (num-bars (get-nth-sequenz (piece mini) 2 'hn 2)))
  ;; insert an rsb in piece=mini, section=2, seq=3 (1-based), player='hn,
  ;; before rsb=1 of the existing seq
  (insert-bar (piece mini) new-bar 1 2 'hn 3 '(1 2 3))
```

```
;; insert an rsb in piece=mini, section=2, seq=3 (1-based), player='vc,
;; before rsb=1 of the existing seq
(insert-bar (piece mini) new-bar 1 2 'vc 3 '(1 2 3))
;; print the number of bars in the sequenz in piece=mini, section=2,
;; seq=2 (0=based), player='hn.
;; now has 2 bars.
(print (num-bars (get-nth-sequenz (piece mini) 2 'hn 2)))
;; update slots of the sc object.
(update-slots mini)
;; print the number of bars of the slippery-chicken object.
;; now 16.
(print (num-bars mini)))
```

```
=>
15
1
2
16
```

SYNOPSIS:

```
(defmethod insert-bar ((p piece) (rsb rthm-seq-bar) bar-num
                       ;; these aren't actually optional but we don't
                       ;; need them in the rthm-seq method
                       &optional section player seq-num ; seq-num is 1-based!
                                 ;; this really is optional
                                 pitch-seq)
```

28.5.10   piece/rebar

[piece] [Methods]

DATE:

 29-Jan-2010

DESCRIPTION:

 Go through the sequences and rebar according to the first one that has the
 least number of bars (but following the player hierarchy).

ARGUMENTS:

 - A piece object (usually provided by calling from the slippery-chicken
   class)

OPTIONAL ARGUMENTS:

- A list of player IDs from the given piece object, ordered in terms of
  importance i.e. which instrument's bar structure should take precedence.

NB: The optional arguments are actually required in this class (not in
    slippery-chicken) but the rebar-fun is not yet used.

RETURN VALUE:

 Always T.

SYNOPSIS:

(defmethod rebar ((p piece) &optional instruments-hierarchy rebar-fun)

28.5.11   bar-holder/player-section

[bar-holder] [Classes]

NAME:

 player-section

 File: player-section.lsp

 Class Hierarchy: named-object -> linked-named-object -> bar-holder
                   -> player-section
                   AND
                   named-object -> linked-named-object -> sclist
                   -> player-section

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of player-section class which is simply a
                   bar holder that contains a list of sequenzes for a
                   particular player.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 18th March 2002

 $$ Last modified: 19:05:48 Fri Feb 2 2018 CET

 SVN ID: Id

28.5.12   bar-holder/section

[bar-holder] [Classes]

NAME:

```
 section

 File:             section.lsp

 Class Hierarchy:  named-object -> linked-named-object -> bar-holder
                   -> section

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of section class which is simply a bar
                   holder and recursive-assoc-list that contains (possibly
                   subsections which contain) player-sections.

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    23rd March 2002

 $$ Last modified:  15:20:01 Fri Jan 10 2020 CET

 SVN ID: $Id$
```

28.5.13   section/get-all-players

[section] [Methods]

DESCRIPTION:

```
 Return a list of the IDs from all players in a section object. NB: When
 retrieving from a section within a slippery-chicken object, all players in
 the ensemble will be returned, as the slippery-chicken object will generate
 rest bars for players even when they're not active in a given section.
```

ARGUMENTS:

```
 - A section object.
```

RETURN VALUE:

 - A list of player IDs (symbols).

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (print (get-all-players (get-section mini 1)))
  (print (get-all-players (get-section mini 2)))
  (print (get-all-players (get-section mini 3))))

=>
(CL HN VC)
(CL HN VC)
(CL HN VC)
```

SYNOPSIS:

```
(defmethod get-all-players ((s section))
```

28.5.14   section/get-bar

[section] [Methods]

DESCRIPTION:

 Return the rthm-seq-bar object at the specified bar number within the given
 section.

NB: The bar number is counted from the beginning of the entire piece, not
    the beginning of the section.

ARGUMENTS:

- A section object.
- An integer that is the bar number for which to return the rthm-seq-bar
  object. This number is 1-based and counts from the beginning of the
  piece, not the beginning of the section.

OPTIONAL ARGUMENTS:

NB: The <player> argument is actually required, but is listed as optional
    for reasons of class inheritance.

- The ID of the player for whose part the rthm-seq-bar object is to be
  returned.

RETURN VALUE:

A rthm-seq-bar object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5)))
                       (2 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5)))
                       (3 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (2 2 2 2 2))
                   (3 (3 3 3 3 3)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (2 ((((4 4) q e s s h))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (3 ((((4 4) e s s h q))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
```

```
                           (2 ((cl (2 2 2 2 2))
                               (hn (2 2 2 2 2))
                               (vc (2 2 2 2 2))))
                           (3 ((cl (3 3 3 3 3))
                               (hn (3 3 3 3 3))
                               (vc (3 3 3 3 3)))))))))
  (get-bar (get-section mini 3) 11 'hn))

=>
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: T, bar-num: 11,
              old-bar-nums: NIL, write-bar-num: NIL, start-time: 40.000,
              start-time-qtrs: 40.0, is-rest-bar: NIL, multi-bar-rest: NIL,
              show-rest: T, notes-needed: 5,
              tuplets: NIL, nudge-factor: 0.35, beams: NIL,
              current-time-sig: 2, write-time-sig: NIL, num-rests: 0,
              num-rhythms: 5, num-score-notes: 5, parent-start-end: NIL,
              missing-duration: NIL, bar-line-type: 0,
              player-section-ref: (3 HN), nth-seq: 0, nth-bar: 0,
              rehearsal-letter: NIL, all-time-sigs: (too long to print)
              sounding-duration: 4.000,
              rhythms: (
[...]
)
SCLIST: sclist-length: 6, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "3-bar1", tag: NIL,
data: ((4 4) E S S H Q)
```

SYNOPSIS:

```
(defmethod get-bar ((s section) bar-num &optional player)
```

28.5.15   section/get-sequenz

[section] [Methods]

DESCRIPTION:

Get the specified sequenz object from a given section object.

ARGUMENTS:

- A section object.
- The ID of the player from whose part the sequenz object is to be
  returned.

- An integer that is the number of the sequence object to be returned from
  within the given section object. This number is 1-based.

RETURN VALUE:

 A sequenz object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (2 ((((4 4) q e s s h))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (3 ((((4 4) e s s h q))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (2 2 2 2 2))
                            (vc (2 2 2 2 2))))
                        (3 ((cl (3 3 3 3 3))
                            (vc (3 3 3 3 3)))))))))
  (get-sequenz (get-section mini 2) 'vc 2))

=>
SEQUENZ: pitch-curve: (1 2 3 4 5)
RTHM-SEQ: num-bars: 1
          num-rhythms: 5
          num-notes: 5
          num-score-notes: 5
          num-rests: 0
          duration: 4.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
```

```
LINKED-NAMED-OBJECT: previous: (1), this: (2), next: (3)
BAR-HOLDER:
            start-bar: 7
            end-bar: 7
            num-bars: 1
            start-time: 24.0
            end-time: 28.0
            start-time-qtrs: 24.0
            end-time-qtrs: 28.0
            num-notes (attacked notes, not tied): 5
            num-score-notes (tied notes counted separately): 5
            num-rests: 0
            duration-qtrs: 4.0
            duration: 4.0 (4.000)
```

SYNOPSIS:

```
(defmethod get-sequenz ((s section) player seq-num) ; 1-based
```

28.5.16   section/has-subsections

[section] [Methods]

DESCRIPTION:

 Boolean test to determine whether a specified section of a slippery-chicken
 object has subsections.

ARGUMENTS:

 - A section object.

RETURN VALUE:

 T if the specified section has subsections, otherwise NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1
                    ((a (1 1 1))
```

```
                          (b (1 1 1))))
                    (2 (1 1 1 1))
                    (3
                     ((a (1 1 1))
                      (b
                        ((x (1 1 1))
                         (y (1 1 1))))))
                    (4
                     ((a (1 1 1))
                      (b (1 1 1))
                      (c (1 1 1 1)))))
          :rthm-seq-palette '((1 ((((2 4) (q) e (s) s))
                                   :pitch-seq-palette ((1 2)))))
          :rthm-seq-map '((1
                            ((a ((vn (1 1 1))))
                             (b ((vn (1 1 1))))))
                           (2 ((vn (1 1 1 1))))
                           (3
                            ((a ((vn (1 1 1))))
                             (b
                               ((x ((vn (1 1 1))))
                                (y ((vn (1 1 1))))))))
                           (4
                            ((a ((vn (1 1 1))))
                             (b ((vn (1 1 1))))
                             (c ((vn (1 1 1 1)))))))))))
  (print (has-subsections (get-section mini 1)))
  (print (has-subsections (get-section mini 2))))

=>
T
NIL
```

SYNOPSIS:

```
(defmethod has-subsections ((s section))
```

28.5.17   section/num-sequenzes

[section] [Methods]

DESCRIPTION:

 Get the number of sequenz objects in a given section object.

ARGUMENTS:

- A section object.

RETURN VALUE:

An integer that is the number of sequenz objects in the specified section
object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vc (cello :midi-channel 1))))
        :set-palette '((1 ((f3 g3 a3 b3 c4))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((vc (1 1 1 1 1))))
                        (2 ((vc (1 1 1 1 1))))
                        (3 ((vc (1 1 1 1 1)))))))))
  (num-sequenzes (get-section mini 2)))

=> 5
```

SYNOPSIS:

```
(defmethod num-sequenzes ((s section))
```

28.5.18   section/re-bar

[section] [Methods]

DESCRIPTION:

Regroup the consecutive note and rest events of a given section object into
new bars of the specified time signature.

This method will only combine short bars into longer ones; it will not
split up longer bars and recombine them.

The method will also use the specified (or default) time signature as a
target, and may be forced to create a number of bars that are not of the
specified time signature if the number of beats in the given section object

do not correspond.

NB: The user must call the update-slots method after using this method as a
    post-generation editing method.

ARGUMENTS:

- A section object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :start-bar. An integer that is the first bar within the specified section
  that is to be re-barred. Default = First bar of the given section.
- :end-bar. An integer that is the last bar within the specified section
  that is to be re-barred. Default = Last bar of the given section.
- :min-time-sig. The target time signature for all new bars. NB: Depending
  on the number of beats in the given section, the method may have to
  deviate from this target time signature. Default = '(2 4).
- :verbose. T or NIL to indicate whether to print feedback on the
  re-barring process to the Lisp listener. Default = NIL.
- :auto-beam. T, NIL, or an integer. If T, the method will automatically
  attach beam indications to the corresponding events according to the beat
  unit of the time signature. If an integer, the method will beam in
  accordance with a beat unit that is equal to that integer. If NIL, the
  method will not automatically place beams. Default = T.

RETURN VALUE:

 T.

EXAMPLE: SYNOPSIS:

```
(defmethod re-bar ((s section)
                   &key start-bar
                   end-bar
                   (min-time-sig '(2 4))
                   verbose
                   ;; could also be a beat rhythmic unit
                   (auto-beam t))
```

28.5.19   bar-holder/sequenz

[bar-holder] [Classes]

NAME:

```
sequenz

File:             sequenz.lsp

Class Hierarchy:  named-object -> linked-named-object -> sclist -> rthm-seq
                  -> sequenz
                                AND
                  named-object -> linked-named-object -> bar-holder
                  -> sequenz

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Implementation of the sequenz class which holds the
                  necessary data (pitch, rhythms etc.) for one sequenz for
                  one instrument.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    March 15th 2002

$$ Last modified:  14:12:35 Tue Aug  4 2020 CEST

SVN ID: $Id$
```

28.5.20   bar-holder/transpose-bars

[bar-holder] [Methods]

DESCRIPTION:

Transpose the pitches of the specified bars in a given player's part by a
specified number of semitones.

ARGUMENTS:

- A bar-holder object (such as the PIECE slot of a slippery-chicken
  object).
- A positive or negative integer that is the number of semitones by which
  the pitches of the specified bars are to be transposed.
- An integer that is the number of the first bar in which the pitches are
  to be transposed.
- An integer that is the number of consecutive bars, including the
  start-bar, in which the pitches are to be transposed.
- The ID of the player whose part is to be changed.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :destructively. T or NIL to indicate whether the transposed pitches are
  to replace the existing pitches of the given bar-holder object. This must
  be set to T if the pitches of the original object are to be transposed
  before, for example, generating output. If NIL, the original object will
  first be cloned, the pitches of the original object will be left
  untouched and the changes will be made to the copy. Default = NIL.
- :print-bar-nums. T or NIL to indicate whether the method should print
  feedback about which bars have been transposed to the listener.
  T = print feedback. Default = NIL.
- :chord-function. The function that is to be used for transposition of
  chords objects. Default = #'transpose (of the chord class).
- :chord-function. The function that is to be used for transposition of
  pitch objects. Default = #'transpose (of the pitch class).
```

RETURN VALUE:

```
 Returns a list of the rthm-seq-bar objects that have been transposed.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((c2 d2 e2 f2 g2 a2 b2
                               c3 d3 e3 f3 g3 a3 b3
                               c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e (s) s))
                                :pitch-seq-palette ((1 (2) 3 4)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (transpose-bars (piece mini) 11 2 2 'cl
                  :destructively t
                  :print-bar-nums t))
```

SYNOPSIS:

```
(defmethod transpose-bars ((bh bar-holder) semitones start-bar num-bars player
                           &key
                           (destructively nil)
```

```
                              (print-bar-nums nil)
                              ;; the default functions are the class methods for
                              ;; pitch or chord.
                              (chord-function #'transpose)
                              (pitch-function #'transpose))
```

28.5.21   linked-named-object/instrument

[linked-named-object] [Classes]

NAME:

```
 instrument

 File:              instrument.lsp

 Class Hierarchy:   named-object -> linked-named-object -> instrument

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           Implementation of the instrument class which defines
                    musical instrument properties like range and
                    collects/stores information about what the instrument
                    plays: how many notes, in how many bars etc.

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     4th September 2001

 $$ Last modified:  17:01:58 Sat Sep 26 2020 CEST

 SVN ID: $Id$
```

28.5.22   instrument/auto-set-subset-id

[instrument] [Methods]

DATE:

```
 July 27th 2018, Heidhausen
```

DESCRIPTION:

```
 Set the subset-id slot to be the same as the id slot.
```

ARGUMENTS:

- an instrument instance

RETURN VALUE:

- the instrument instance

SYNOPSIS:

```
(defmethod auto-set-subset-id ((ins instrument))
```

28.5.23 instrument/default-chord-function

[instrument] [Functions]

DESCRIPTION:

If an instrument is able to play chords, a function must be defined to
select pitches from a list that it can play as a chord. This function (as a
symbol) is passed as a slot to the instrument instance.

This is the default function. It returns a 2-note chord with the pitch at
index plus that below it, or that above it if there are no lower pitches
available. Or it just returns a single-pitch chord object if neither of
those cases are possible.

NB: The arguments are supplied by slippery chicken when it calls the
    function.

ARGUMENTS:

- The current number from the pitch-seq. Currently ignored by default.
- The index that the first argument was translated into by the offset and
  scaler (based on trying to get a best fit for the instrument and set).
  This can be assumed to be a legal reference into pitch-list as it was
  calculated as fitting in pitch-seq::get-notes. (zero-based.)
- The pitch-list created from the set, taking into account the instrument's
  range and other notes already played by other instruments.
- The current pitch-seq object. Currently ignored by default.
- The current instrument object. Currently ignored by default.
- The current set object. Currently ignored by default.

RETURN VALUE:

 A chord object.

SYNOPSIS:

```
(defun default-chord-function (curve-num index pitch-list pitch-seq
                                instrument set)
```

28.5.24   instrument/force-in-range

[instrument] [Methods]

DATE:

 October 9th 2013

DESCRIPTION:

 Forces a pitch to be within an instrument's range by transposing up or down
 the required number of octaves. NB it doesn't force a microtonal pitch to
 the nearest chromatic pitch so whether it's playable on an instrument is not
 checked here.

ARGUMENTS:

 - the instrument object
 - the piece object

OPTIONAL ARGUMENTS:

 keyword argument:
 - :sounding. Whether the pitch should be considered a sounding pitch.
   Default = NIL.

RETURN VALUE:

 A pitch object within the instrument's range.

EXAMPLE:

```
(let ((cl (get-standard-ins 'b-flat-clarinet)))
  ;; needs to go down 1 octave
  (print (data (force-in-range cl (make-pitch 'e7))))
  ;; needs to go up 2 octaves
  (print (data (force-in-range cl (make-pitch 'g1))))
```

```
;; the t indicates we're dealing with sounding pitches so here there's no
;; transposition...
(print (data (force-in-range cl (make-pitch 'd3) :sounding t)))
;; ... but here there is
(print (data (force-in-range cl (make-pitch 'd3)))))
=>
E6
G3
D3
D4
```

SYNOPSIS:

```
(defmethod force-in-range ((ins instrument) pitch &key sounding)
```

28.5.25   instrument/in-range

[instrument] [Methods]

DESCRIPTION:

Checks whether a specified pitch/chord falls within the defined range of a
given instrument object or not.

ARGUMENTS:

- An instrument object.
- A pitch or chord item (either a chord object, pitch object, or note-name
  symbol). Chords are handled by a separate method (of the same name).

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the pitch specified is to be compared with
  the given instrument object's sounding or written range. T = Sounding.
  Default = NIL. If T, a secondary NIL is also returned to indicate that
  the specified pitch is neither too high nor too low.
- T or NIL to indiciate whether an artificial harmonic should be tried if
  the pitch is too high for an instrument whose :harmonics slot is
  T. Default = NIL.
- T or NIL to indicate whether a microtonal pitch should be considered
  'in-range' when the instrument can't play microtones. Default = T (as
  theoretically it's within the range even though not playable).
- T or NIL to indicate whether pitches the instrument can't play (the
  missing-notes slot) should cause the method to return NIL. Default = NIL =
  even 'missing notes' will be considered in range.

RETURN VALUE:

```
Returns T if the specified pitch falls between the
lowest-sounding/lowest-written and the highest-sounding/highest-written
pitches of the given pitch object, otherwise NIL.

If the specified pitch is outside of the range, an additional value of 0 or
1 is also returned to indicate whether the specified pitch is too high (1)
or too low (0). If it's out of range but could be achieved by an artificial
harmonic (just the perfect 4th variety) on an instrument with :harmonics T,
then the additional value will be a chord object instead (the two pitches,
including diamond notehead for the second, which would achieve the desired
pitch.
```

EXAMPLE:

```
;; Determine if a pitch provided as a note-name symbol falls within the written
;; range of a non-transposing instrument
(let ((i1 (make-instrument 'inst1 :lowest-written 'bf3 :highest-written 'a6)))
  (in-range i1 'c4))
  => T, NIL

;; Determine if a pitch provided as a note-name symbol falls within the
;; sounding range of a transposing instrument, using the optional argument T
(let ((i2 (make-instrument 'inst1 :lowest-written 'fs3 :highest-written 'c6
                          :transposition 'BF)))
  (in-range i2 'c6 T))
  => NIL, 1

;; A pitch object can be used as the specified pitch
(let ((i2 (make-instrument 'inst1 :lowest-written 'fs3 :highest-written 'c6
                          :transposition 'BF)))
  (in-range i2 (make-pitch 'd6)))
  => NIL, 1
```

SYNOPSIS:

```
(defmethod in-range ((ins instrument) pitch
                    &optional sounding try-artificial-harmonic
                      (impossible-microtones t) no-missing-notes)
```

28.5.26   instrument/make-instrument

[instrument] [Functions]

DESCRIPTION:

Create an instrument object, specifying the values for a number of
parameters for describing characteristics of a given instrument, such as
lowest and highest pitch, transposition, clefs used by the instrument etc.

NB: The user will generally define instruments in the context of
the :instrument-palette slot of a call to make-slippery-chicken, or an
explicit call to the make-instrument-palette function.

ARGUMENTS:

- A symbol that is the instrument ID.

OPTIONAL ARGUMENTS:

keyword arguments:
- :staff-name. String. This is the unabbreviated instrument name that will
  be used for the first page of printed scores.
- :staff-short-name. String. This is the abbreviated instrument name that
  will be used for subsequent pages of printed scores.
- :lowest-written. Note-name symbol. This is the lowest written pitch
  available on the given instrument. Defaults to NIL. A user may only
  define either the lowest-written value or the lowest-sounding value. If a
  lowest-written value is given, the method automatically determines the
  lowest-sounding value based on the lowest-written value and the
  transposition value.
- :highest-written. Note-name symbol. This is the highest written pitch
  available on the given instrument. Defaults to NIL. A user may only
  define either the highest-written value or the highest-sounding value. If
  a highest-written value is given, the method automatically determines the
  highest-sounding value based on the highest-written value and the
  transposition value.
- :lowest-sounding. Note-name symbol. This is the lowest sounding pitch
  available on the given instrument. Defaults to NIL. A user may only
  define either the lowest-sounding value or the lowest-written value. If a
  lowest-sounding value is given, the method automatically determines the
  lowest-written value based on the lowest-sounding value and the
  transposition value.
- :highest-sounding. Note-name symbol. This is the highest sounding pitch
  available on the given instrument. Defaults to NIL. A user may only
  define either the highest-sounding value or the highest-written value. If
  a highest-sounding value is given, the method automatically determines
  the highest-written value based on the highest-sounding value and the
  transposition value.
- :transposition. Note-name symbol. This is the key of the given instrument
  (such as the "B-flat" of the "B-flat clarinet"), given as a note-name
  symbol (such as 'BF for B-flat). If a value is only given for the

:transposition argument but not for the :transposition-semitones
argument, and there are multiple semitone transposition options for the
key specified, the method will choose the most common semitone
transposition for that given key. NB: When using keyword argument
:transposition rather than :transposition-semitones, sc will have a
warning printed by cm with indications as to which direction the
transposition has been undertaken.
- :transposition-semitones. Integer (positive or negative). The number of
  semitones lower that a given instrument sounds than written, e.g. -2 for
  B-flat Clarinet. If a value is only given for the
  :transposition-semitones argument but not for the :transposition
  argument, the method will automatically determine the key for the
  :transposition argument. The listener will drop into the debugger with an
  error if a key is given for the :transposition argument and the number
  specified for the :transposition-semitones does not correspond with that
  key.
- :starting-clef. Symbol. This value determines the first clef that a given
  instrument is to use if that instrument can use different clefs. For a
  list of available clefs see the :clefs argument below.
  Default = 'treble.
- :clefs. List of symbols. All clefs that a given instrument may use in the
  course of a piece. Clefs available are treble, treble-8vb, alto, tenor,
  bass, percussion, double-treble, and double-bass. Clefs are to be given in
  order of preference. Defaults automatically to the value given to
  :starting-clef if no other clefs are specified. NB: If a separate list is
  indeed given here, the method will automatically add the value for
  :starting-clef as well, should it have been omitted. In this case, a
  warning will also be printed.
- :clefs-in-c. List of symbols. Similar to :clefs, but designates which
  clefs an instrument uses in a C-score; for example, bass clarinet may
  notated in bass clef for sounding pitches though it is standardly
  notated in treble clef for written pitches. For a list of clefs available
  see the :clefs argument above.
- :largest-fast-leap. Number. This value indicates the largest interval, in
  semitones, that a player can feasibly perform at a fast tempo on the
  given instrument. Default = 999. "Fast" here is determined for the whole
  piece by the slippery-chicken class's fast-leap-threshold slot.
- :score-write-in-c. T or NIL. Determines whether the musical material for
  the given instrument should be printed in C. T = print in C.
  Default = NIL.
- :score-write-bar-line. Integer. This argument is used for indicating
  system-grouping in the printed score. The given integer specifies how
  many instruments above this one should be grouped together with an
  unbroken bar-line. Default = 1.
- :midi-program. Integer. The number of the MIDI program to be used for
  playing back this instrument. Default = 1.

- :chords. T or NIL. Indicates whether the given instrument is capable of
  playing chords (starting with 2-note simultaneities, but not
  multiphonics).
- :subset-id. Symbol, string, number, or NIL. Indicates the ID of a
  specific subset of the current set to which the instrument's pitch
  selection is limited. No error will occur if no subset with this ID
  exists in a given set, i.e. some may include this subset, some may not
  and everything will function correctly--if the specified subset is not
  present in the current set the pitch selection routine will select from
  the whole set. In every case however, the usual set limiting according
  to instrument range etc. will also apply. Default = NIL.
- :microtones. T or NIL. Indicates whether the instrument can play
  microtones. T = can play microtones. Default = NIL. NB: If this value is
  set to T, a separate :microtones-midi-channel must be specified; this can
  be done for the given instrument object in the :ensemble block of the
  make-slippery-chicken function.
- :missing-notes. A list of note-name symbols. This is a list of any notes
  which the given instrument can't play, for example certain
  quarter-tones. These are to be given by the user as written-pitch
  note-name symbols, but are always stored by the method as sounding
  pitches.
- :prefers-notes. Symbol. 'high, 'low or NIL. This value indicates whether
  to give preference, when choosing notes for the given instrument, to
  pitches from the upper or lower end of the instrument's range. When NIL,
  preference is given to notes from its middle register. Default = NIL.
- :chord-function. If the given instrument can play chords then it will
  need a reference to a function that can select chords for it. NB This
  should be a symbol not a function object; thus, 'my-fun not
  #'my-fun. Default = NIL.
- :staff-lines. How many the lines the staff should have when this
  instrument is playing. When we change instrument mid-piece the number of
  staff lines will change accordingly. Similarly, if we start with an
  instrument that doesn't use 5 staff lines, we'll begin a piece with the
  correct number. Default = 5.

RETURN VALUE:

 Returns an instrument object.

EXAMPLE:

```
;; Make-instrument for the flute:         ;
  (make-instrument 'flute :staff-name "Flute" :staff-short-name "Fl."
:lowest-written 'c4 :highest-written 'd7
:starting-clef 'treble :midi-program 74 :chords nil
:microtones t :missing-notes '(cqs4 dqf4))
```

```
 =>
 INSTRUMENT: lowest-written:
 PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
 [...]
 , highest-written:
 PITCH: frequency: 2349.318, midi-note: 98, midi-channel: 0
 [...]
 lowest-sounding:
 PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
 [...]
 , highest-sounding:
 PITCH: frequency: 2349.318, midi-note: 98, midi-channel: 0
 starting-clef: TREBLE, clefs: (TREBLE), clefs-in-c: (TREBLE)
 prefers-notes: NIL, midi-program: 74
 transposition: C, transposition-semitones: 0
 score-write-in-c: NIL, score-write-bar-line: 1
 chords: NIL, chord-function: NIL,
 total-bars: 0 total-notes: 0, total-duration: 0.0
 total-degrees: 0, microtones: T
 missing-notes: (CQS4 DQF4), subset-id: NIL
 staff-name: Flute, staff-short-name : Fl.,
 largest-fast-leap: 999
 [...]
 NAMED-OBJECT: id: FLUTE, tag: NIL,
 data: NIL

;; A make-instrument for the b-flat bass clarinet ;
 (make-instrument 'bass-clarinet :staff-name "Bass Clarinet" :lowest-written 'c3
:highest-written 'g6 :staff-short-name "Bass Cl."
:chords nil :midi-program 72 :starting-clef 'treble
:microtones t :prefers-notes 'low
:missing-notes '(aqs4 bqf4 bqs4 cqs5 dqf5 gqf3 fqs3 fqf3 eqf3
dqs3 dqf3 cqs3)
:clefs '(treble) :clefs-in-c '(treble bass)
:transposition-semitones -14)

 =>
 INSTRUMENT: lowest-written:
 PITCH: frequency: 130.813, midi-note: 48, midi-channel: 0
 [...]
 , highest-written:
 PITCH: frequency: 1567.982, midi-note: 91, midi-channel: 0
 [...]
 lowest-sounding:
 PITCH: frequency: 58.270, midi-note: 34, midi-channel: 0
```

```
[...]
, highest-sounding:
PITCH: frequency: 698.456, midi-note: 77, midi-channel: 0
[...]
NAMED-OBJECT: id: BASS-CLARINET, tag: NIL,
data: NIL
```

SYNOPSIS:

```
(defun make-instrument (id &key
                            staff-name
                            staff-short-name
                            lowest-written
                            highest-written
                            lowest-sounding
                            highest-sounding
                            transposition
                            transposition-semitones
                            (starting-clef 'treble)
                            clefs
                            (largest-fast-leap 999)
                            score-write-in-c
                            (score-write-bar-line 1)
                            (midi-program 1)
                            (staff-lines 5)
                            chords
                            clefs-in-c
                            subset-id
                            microtones
                            missing-notes
                            prefers-notes
                            chord-function harmonics open-strings
                            open-string-marks
                            (nodes '((12 2) (7 3) (5 4) (4 5)))))
```

28.5.27   instrument/natural-harmonic?

[instrument] [Methods]

DATE:

 8th January 2019, Heidhausen

DESCRIPTION:

 Determine whether a note can be played as a natural harmonic. If so, then we

return the appropriate pitch object with harmonic marks, etc. (see below).

In order for this to work, the :harmonics slot of the instrument
will need to be T and the open-strings, nodes, and open-string-marks slots
must be set appropriately. The default nodes are ((12 2) (7 3) (5 4) (4 5)
(3 6)) where the first of each pair is the nodal point in semitones above
the open string pitch and the second is the partial number produced. The
open-strings slot should be a list of descending pitches (symbols or
objects) and the open-strings-marks is something like (i ii iii iv) for
orchestral strings or (c1 c2 c3 c4 c5 c6) for guitar. Note that these two
slots must be in the same order and also that the marks must be recognised
by the output score routine if they are to be meaningful.

NB The note returned is the 'written' note (in the case of e.g. double-bass)
but won't be transposed down an octave for e.g. harp notation.

See also the natural-harmonic function in pitch.lsp for an on-the-fly (and
thus less efficient) routine that works with any arbitrary string tuning
independently of instrument objects.

ARGUMENTS:

 - an instrument object
 - a note, either as a pitch symbol or object. This should be the written
   pitch in the case of transposing instruments such as the guitar or double
   bass.

OPTIONAL ARGUMENTS:

 - frequency tolerance in cents. (A natural major third (5th partial) is 14
   cents from its nearest equal-tempered neighbour.) Default = 15.

RETURN VALUE:

A pitch object with marks attached for the harmonic (circle) and string,
where appropriate, plus, as a second value, and only for instruments with
open-strings (e.g. orchestral strings) the note at the nodal point, with a
diamond note-head mark. These could be combined into a chord for notation
purposes.

EXAMPLE:

```
(natural-harmonic? (get-standard-ins 'violin) 'bf5)
-->
NIL
```

```
NIL

(natural-harmonic? (get-standard-ins 'violin) 'b5)
-->
PITCH: frequency: 987.767, midi-note: 83, midi-channel: 1
       pitch-bend: 0.0
       degree: 166, data-consistent: T, white-note: B5
       nearest-chromatic: B5
       src: 3.7754972, src-ref-pitch: C4, score-note: B5
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 5, c5ths: 0, no-8ve: B, no-8ve-no-acc: B
       show-accidental: T, white-degree: 48,
       accidental: N,
       accidental-in-parentheses: NIL, marks: (IV HARM),
...

;;; or with scordatura:
(let ((vln (get-standard-ins 'violin)))
  (setf (open-strings vln) '(eqs5 a4 d4 g3))
  (prog1
      (natural-harmonic? vln 'eqs6)
    (setf (open-strings vln) '(e5 a4 d4 g3)))) ; reset
-->
PITCH: frequency: 1357.146, midi-note: 88, midi-channel: 1
       pitch-bend: 0.5
       degree: 177, data-consistent: T, white-note: E6
       nearest-chromatic: E6
       src: 5.1873584, src-ref-pitch: C4, score-note: ES6
       qtr-sharp: 1, qtr-flat: NIL, qtr-tone: 1,
       micro-tone: T,
       sharp: NIL, flat: NIL, natural: NIL,
       octave: 6, c5ths: 0, no-8ve: EQS, no-8ve-no-acc: E
       show-accidental: T, white-degree: 51,
       accidental: QS,
       accidental-in-parentheses: NIL, marks: (I HARM),
...
```

SYNOPSIS:

```
(defmethod natural-harmonic? ((ins instrument) note &optional (tolerance 15))
```

28.5.28   instrument/prefers-high

[instrument] [Methods]

DESCRIPTION:

 Determine whether the PREFERS-NOTES slot of a given instrument object is
 set to 'HIGH.

ARGUMENTS:

 - An instrument object.

RETURN VALUE:

 Returns T if the PREFERS-NOTES slot of the given instrument object is set
 to 'HIGH, otherwise NIL.

EXAMPLE:

```
;; Returns T if the PREFERS-NOTES slot of the given instrument object is set to ;
;; 'HIGH                                    ;
  (let ((i1 (make-instrument 'inst :prefers-notes 'high)))
(prefers-high i1))

  => T

;; Returns NIL if the PREFERS-NOTES slot of the given instrument object is not ;
;; set to 'HIGH                            ;
  (let ((i1 (make-instrument 'inst1))
(i2 (make-instrument 'inst2 :prefers-notes 'low)))
(print (prefers-high i1))
(print (prefers-high i2)))

  =>
 NIL
 NIL
```

SYNOPSIS:

```
(defmethod prefers-high ((ins instrument))
```

28.5.29   instrument/prefers-low

[instrument] [Methods]

DESCRIPTION:

Determine whether the PREFERS-NOTES slot of a given instrument object is
set to 'LOW.

ARGUMENTS:

 - An instrument object.

RETURN VALUE:

 Returns T if the PREFERS-NOTES slot of the given instrument object is set
 to 'LOW, otherwise NIL.

EXAMPLE:

```
;; Returns T if the PREFERS-NOTES slot of the given instrument object is set to ;
;; 'LOW                                 ;
  (let ((i1 (make-instrument 'inst :prefers-notes 'low)))
(prefers-low i1))

  => T

;; Returns NIL if the PREFERS-NOTES slot of the given instrument object is not ;
;; set to 'LOW                          ;
  (let ((i1 (make-instrument 'inst1))
(i2 (make-instrument 'inst2 :prefers-notes 'high)))
(print (prefers-low i1))
(print (prefers-low i2)))

  =>
 NIL
 NIL
```

SYNOPSIS:

```
(defmethod prefers-low ((ins instrument))
```

28.5.30   instrument/set-prefers-high

[instrument] [Methods]

DATE:

 05 Feb 2011

DESCRIPTION:

Sets the PREFERS-NOTES slot of the given instrument object to 'HIGH.

ARGUMENTS:

- An instrument object.

OPTIONAL ARGUMENTS:

(- optional ignore argument; for internal use only).

RETURN VALUE:

Returns symbol HIGH.

EXAMPLE:

```
;; Returns symbol HIGH by default        ;
  (let ((i1 (make-instrument 'inst)))
(set-prefers-high i1))

  => HIGH

;; Create an instrument object with only an ID, print the PREFERS-NOTES slot to ;
;; see that it is NIL by default, apply the set-prefers-high, and print the ;
;; slot again to see the changes         ;
  (let ((i1 (make-instrument 'inst)))
(print (prefers-notes i1))
(set-prefers-high i1)
(print (prefers-notes i1)))

  =>
  NIL
  HIGH

;; Reset to HIGH from LOW                 ;
  (let ((i1 (make-instrument 'inst :prefers-notes 'low)))
(print (prefers-notes i1))
(set-prefers-high i1)
(print (prefers-notes i1)))

  =>
  LOW
  HIGH
```

SYNOPSIS:

```
(defmethod set-prefers-high ((ins instrument) &optional ignore)
```

28.5.31   instrument/set-prefers-low

[instrument] [Methods]

DATE:

 05 Feb 2011

DESCRIPTION:

 Sets the PREFERS-NOTES slot of the given instrument object to 'LOW.

ARGUMENTS:

 - An instrument object.

OPTIONAL ARGUMENTS:

 (- optional ignore argument; for internal use only).

RETURN VALUE:

 Returns symbol LOW.

EXAMPLE:

```
;; Returns symbol LOW by default
  (let ((i1 (make-instrument 'inst)))
(set-prefers-low i1))

  => LOW

;; Create an instrument object with only an ID, print the PREFERS-NOTES slot to
;; see that it is NIL by default, apply the set-prefers-low, and print the
;; slot again to see the changes
  (let ((i1 (make-instrument 'inst)))
(print (prefers-notes i1))
(set-prefers-low i1)
(print (prefers-notes i1)))

  =>
  NIL
  LOW

;; Reset to LOW from HIGH
```

```
  (let ((i1 (make-instrument 'inst :prefers-notes 'high)))
(print (prefers-notes i1))
(set-prefers-low i1)
(print (prefers-notes i1)))

  =>
  HIGH
  LOW
```

SYNOPSIS:

```
(defmethod set-prefers-low ((ins instrument) &optional ignore)
```

28.5.32   instrument/transposing-instrument-p

[instrument] [Methods]

DESCRIPTION:

 Determine whether a given instrument object defines a transposing
 instrument.

ARGUMENTS:

 - An instrument object.

OPTIONAL ARGUMENTS:

 - ignore-octaves. T or NIL to indicate whether instruments that transpose
   at the octave are to be considered transposing instruments.
   T = instruments that transpose at the octave are not considered
   transposing instruments. Default = T.

RETURN VALUE:

 Returns T if the given instrument object defines a transposing instrument,
 otherwise NIL.

EXAMPLE:

```
;; Returns NIL if the instrument is not a transposing instrument ;
  (let ((i1 (make-instrument 'instrument-one)))
(transposing-instrument-p i1))
```

```
  => NIL
```

```
;; Returns T if the instrument object has been defined using a non-NIL value ;
;; for :transposition                    ;
  (let ((i2 (make-instrument 'instrument-two :transposition 'bf)))
(transposing-instrument-p i2))
```

```
  => T
```

```
;; Returns T if the instrument object has been defined using a non-0 value for ;
;; :transposition-semitones              ;
  (let ((i3 (make-instrument 'instrument-two :transposition-semitones -3)))
(transposing-instrument-p i3))
```

```
  => T
```

```
;; Setting the optional argument to NIL causes instruments that transpose at ;
;; the octave to return T.               ;
  (let ((i3 (make-instrument 'instrument-two :transposition-semitones -12)))
(transposing-instrument-p i3))
```

```
  => NIL
```

```
  (let ((i3 (make-instrument 'instrument-two :transposition-semitones -12)))
(transposing-instrument-p i3 nil))
```

```
  => T
```

SYNOPSIS:

```
(defmethod transposing-instrument-p ((ins instrument)
                                     &optional (ignore-octaves t))
```

28.5.33 linked-named-object/pitch

[linked-named-object] [Classes]

NAME:

```
 pitch
```

```
 File:           pitch.lsp
```

```
 Class Hierarchy:  named-object -> linked-named-object -> pitch
```

```
Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Implementation of the pitch class for holding pitch
                  information: symbolic representation (eg c4), MIDI note
                  number, frequency, sampling-rate conversion etc.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    March 18th 2001

$$ Last modified:  15:57:34 Tue Oct 27 2020 CET

SVN ID: $Id$
```

28.5.34   pitch/add-mark

[pitch] [Methods]

DESCRIPTION:

Add a specified mark to the MARKS slot of the given pitch object.

NB: The add-mark method does not check first to see whether the mark being
    added is a legitimate mark. It does print a warning, however, when the
    specified mark is already present in the MARKS slot, though it adds it
    anyway.

ARGUMENTS:

 - A pitch object.
 - A symbol that is a mark.

RETURN VALUE:

A list. The method returns the entire contents of the given pitch object's
MARKS slot as a list.

Prints a warning when the specified mark is already present in the given
pitch object's MARKs slot.

EXAMPLE:

```
;; By default the MARKS slot of a newly created pitch object is set to NIL
```

```
(let ((p (make-pitch 'c4)))
  (marks p))

=> NIL

;; Add two marks and print the contents of the given pitch object's MARKS slot
;; to see the changes
(let ((p (make-pitch 'c4)))
  (add-mark p 'pizz)
  (add-mark p 'a)
  (print (marks p)))

=>
(A PIZZ)

;; Prints a warning when the specified mark is already present in the MARKS
;; slot, though it adds it again anyway.
(let ((p (make-pitch 'c4)))
  (add-mark p 'pizz)
  (add-mark p 'pizz)
  (marks p))

=> (PIZZ PIZZ)
WARNING:
pitch::add-mark: mark PIZZ already present but adding again!
```

SYNOPSIS:

```
(defmethod add-mark ((p pitch) mark &optional warn-rest warn-again)
```

28.5.35   pitch/cents-hertz

[pitch] [Methods]

DATE:

 December 24th 2013

DESCRIPTION:

 Convert an offset in cents into the frequency deviation of a pitch.

ARGUMENTS:

 - a pitch object

- the number of cents to offset the pitch and return the frequency
  deviation for

RETURN VALUE:

 The frequency deviation in Hertz of the offset pitch.

EXAMPLE:

```
;;; taking as a given the usual floating point round errors:
(cents-hertz (make-pitch 'a4) 1200)
=> 439.9999694824219d0 ; i.e. going up an octave from a4 would be 440 Hz higher
(cents-hertz (make-pitch 'a4) -1200)
=> -219.99998474121094d0
(cents-hertz (make-pitch 'a4) 3)
=> 0.7631253666877456d0
(cents-hertz (make-pitch 'a4) 10)
=> 2.5489090105293144d0
(cents-hertz (make-pitch 'a3) 10)
=> 1.2744545052646572d0
```

SYNOPSIS:

```
(defmethod cents-hertz ((p pitch) cents)
```

28.5.36 pitch/cmn-display-pitch-list

[pitch] [Functions]

DESCRIPTION:

 Use CMN to display a list of pitch objects.

ARGUMENTS:

 The list of pitch objects. This could also contain cmn objects if
 needed, e.g. cmn::bar, cmn::interior-double-bar, cmn::line-mark, etc.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :staff. The CMN staff object to display with.  Default = cmn::treble.
- :size.  The CMN size for the staff.  Default = 20.
- :file.  The path of the file to (over)write.
```

```
Default = "pitches.eps" in the directory (get-sc-config 'default-dir)
(default /tmp)
```
 - :auto-open. Whether to open the .EPS file once written. Currently only
   available on OSX with SBCL and CCL. Uses the default app for .EPS
   files, as if opened with 'open' in the terminal. Default = Value of
   (get-sc-config cmn-display-auto-open).

RETURN VALUE:

 A CMN score object.

SYNOPSIS:

```
(defun cmn-display-pitch-list
    (pitches &key (staff cmn::treble) (size 20)
     (auto-open (get-sc-config 'cmn-display-auto-open))
     (file (concatenate 'string (get-sc-config 'default-dir) "pitches.eps")))
```

28.5.37   pitch/degree-

[pitch] [Methods]

DESCRIPTION:

 Determine the difference between the quarter-tone degree of one pitch
 object and that of a second.

 NB: This method does not return absolute difference; instead, it may return
     positive or negative results depending on the order in which the pitch
     objects are given. (This will aid in revealing directionality.)

 NB: The DEGREE slot is measured in quarter-tones, not semitones. Thus,
     middle-C is degree 120, not 60, and the difference between two
     consecutive semitones is 2, not 1.

ARGUMENTS:

 - A first pitch object.
 - A second pitch object.

RETURN VALUE:

 Returns a number. The number may be positive or negative.

EXAMPLE:

```
;; Subtracting the lower pitch object from the higher returns a positive number
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (degree- p1 p2))

=> 4

;; Reversing the order in which the pitch objects are entered may return a
;; negative number
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (degree- p2 p1))

=> -4
```

SYNOPSIS:

```
(defmethod degree- ((p1 pitch) (p2 pitch))
```

28.5.38   pitch/delete-marks

[pitch] [Methods]

DESCRIPTION:

Delete all marks stored in the MARKS slot of the given pitch object and
reset the slot to NIL.

ARGUMENTS:

 - A pitch object.

RETURN VALUE:

 Always returns NIL

EXAMPLE:

```
;; Add two marks, then delete them. The method returns NIL
(let ((p (make-pitch 'c4)))
  (add-mark p 'pizz)
  (add-mark p 'a)
  (delete-marks p))

=> NIL
```

```
;; Add two marks and print the MARKS slot to see the changes. Then apply the
;; delete-marks method and print the MARKS slot to see the changes.
(let ((p (make-pitch 'c4)))
  (add-mark p 'pizz)
  (add-mark p 'a)
  (print (marks p))
  (delete-marks p)
  (print (marks p)))

=>
(A PIZZ)
NIL
```

SYNOPSIS:

```
(defmethod delete-marks ((p pitch))
```

28.5.39   pitch/enharmonic

[pitch] [Methods]

DESCRIPTION:

 Get the enharmonic equivalent of the given pitch object. Two chromatically
 consecutive "white-note" pitches (e.g. B-sharp/C-natural) are considered
 enharmonically equivalent. If there is no enharmonic equivalent, the
 method just returns the same note.

ARGUMENTS:

 - A pitch object.

OPTIONAL ARGUMENTS:

 - T or NIL to print a warning when no enharmonic can be found. Default = T.

RETURN VALUE:

 A new pitch object i.e. this method is non-destructive.

EXAMPLE:

```
;; A "black-key" enharmonic equivalent
```

```
(let ((p (make-pitch 'cs4)))
  (data (enharmonic p)))

=> DF4

;; Two chromatically consecutive "white-keys" are enharmonically equivalent
(let ((p (make-pitch 'f4)))
  (data (enharmonic p)))

=> ES4

;; The method returns a pitch object with the same pitch value if there is no
;; enharmonic equivalent
(let ((p (make-pitch 'g4)))
  (data (enharmonic p)))

=> G4
```

SYNOPSIS:

```
(defmethod enharmonic ((p pitch) &key (warn t))
```

28.5.40   pitch/enharmonic-equivalents

[pitch] [Methods]

DATE:

 25th December 2013

DESCRIPTION:

 Test whether two pitches are enharmonically equivalent.

ARGUMENTS:

```
 - pitch object 1
 - pitch object 2
```

RETURN VALUE:

 T or NIL

EXAMPLE:

```
(enharmonic-equivalents (make-pitch 'gs4) (make-pitch 'af4))
=> T
(enharmonic-equivalents (make-pitch 'gs4) (make-pitch 'gs4))
=> NIL
```

SYNOPSIS:

```
(defmethod enharmonic-equivalents ((p1 pitch) (p2 pitch)))
```

28.5.41   pitch/find-nearest-pitch

[pitch] [Functions]

DATE:

 April 23rd 2016, Edinburgh

DESCRIPTION:

 In a given list of pitches, find the nearest pitch to the pitch given as
 the 2nd argument.

ARGUMENTS:

 - a list of pitches: either pitch objects or symbols. If symbols they will
   be convered to pitch objects first and optionally sorted from low to
   high.
 - the pitch to use in the search for the nearest. Again: either a pitch
   object or symbol.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the pitch list should be sorted from low to
   high before being searched. Default = NIL.

RETURN VALUE:

 Two values: the pitch from the pitch list which is the nearest to the 2nd
 argument, and the index of this pitch in the list. NB if the 2nd argument
 is equidistant from two pitches in the list, then the first pitch in the
 list will be returned.

EXAMPLE:

```
(find-nearest-pitch '(b0 d1 fs1 a4) 'e1)
==>
PITCH: frequency: 36.708, midi-note: 26, midi-channel: 0
       pitch-bend: 0.0
       degree: 52, data-consistent: T, white-note: D1
       nearest-chromatic: D1
       src: 0.14030775, src-ref-pitch: C4, score-note: D1
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 1, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
       show-accidental: T, white-degree: 15,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL,
       marks-before: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: D1, tag: NIL,
data: D1
**************

1
```

SYNOPSIS:

```
(defun find-nearest-pitch (pitch-list pitch &optional auto-sort)
```

28.5.42   pitch/in-octave

[pitch] [Functions]

DESCRIPTION:

Test to see if a specified pitch item falls within a specified octave. The
pitch item can be a pitch object, a numerical frequency value, or a
note-name symbol.

ARGUMENTS:

- A pitch item. This can be a pitch object, a numerical frequency value, or
  a note-name symbol.
- A number that is the specified octave designator (e.g. the "4" in "C4").

RETURN VALUE:

T if the specified pitch item falls within the specified octave, otherwise
NIL.

EXAMPLE:

```
;; The function returns NIL if the specified pitch item does not fall within
;; the specified octave.
(let ((p (make-pitch 'c4)))
  (in-octave p 3))

=> NIL

;; The function will accept pitch objects
(let ((p (make-pitch 'c4)))
  (in-octave p 4))

=> T

;; The function will accept numerical frequency values
(let ((p 261.63))
  (in-octave p 4))

=> T

;; The function will accept note-name symbols
(let ((p 'c4))
  (in-octave p 4))

=> T
```

SYNOPSIS:

```
(defun in-octave (pitch octave)
```

28.5.43   pitch/invert-pitch-list

[pitch] [Functions]

DESCRIPTION:

Using the lowest note in the list as the reference point, invert the rest
of a given list of pitch items according to their distance from it.

The list of pitch items may consist of either note-name symbols, pitch
objects or frequency numbers.

NB: This function adheres to a concept of inversion more related to
    interval set theory than to the traditional inversion of a melodic

contour. The given list of pitch items is first sorted from low to high
before the internal semitone intervals are assessed. The resulting list
will therefore always be in chromatic order, rather than having the
inverted melodic contour of the original.

ARGUMENTS:

 - A list of pitch items. This may consist of pitch objects, note-name
   symbols, or frequency numbers.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the result should be a list of pitch objects
   or a list of note-name symbols. T = note-name symbols. Default = NIL.
 - The package in which the process is to be performed. Default = :sc.

RETURN VALUE:

 Returns list of pitch objects by default. If the first optional argument is
 set to T, the function will return a list of note-name symbols instead.

EXAMPLE:

```
;; Simple example returning pitch symbols:
(invert-pitch-list '(c4 cs4 f4 b4) t)
=>
(C4 B3 G3 CS3)

;; The function returns a list of pitch objects by default
(let ((pl))
  (setf pl (loop for m in '(E4 G4 A4 C4) collect (make-pitch m)))
  (invert-pitch-list pl))

=>
(
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
[...]
PITCH: frequency: 207.652, midi-note: 56, midi-channel: 0
[...]
data: AF3
[...]
PITCH: frequency: 174.614, midi-note: 53, midi-channel: 0
[...]
```

```
data: F3
[...]
PITCH: frequency: 155.563, midi-note: 51, midi-channel: 0
[...]
data: EF3
)
```

```
;; Setting the first optional argument to T will cause the function to return a
;; list of note-name symbols instead
(let ((pl))
  (setf pl '(329.63 392.00 440.00 261.63))
  (invert-pitch-list pl t))
```

```
=> (C4 AF3 F3 EF3)
```

SYNOPSIS:

```
(defun invert-pitch-list (pitch-list &optional
                          (return-symbols nil)
                          (package :sc))
```

28.5.44   pitch/make-pitch

[pitch] [Functions]

DESCRIPTION:

Create a pitch object, specifying a note as either a symbol or a
number. When the note is specified as a symbol, it is treated as a
note-name; when it is specified as a number, it is treated as a frequency
in hertz.

NB If a pitch object is created from a frequency (rather than note symbol)
   then the given frequency is stored and the note/midi-note etc. nearest
   to it will be stored also. So the frequency might not be the exact
   frequency of the reflected note. This is by design, so that unusual
   temperaments can retain exact frequencies and show nearest notes etc.

ARGUMENTS:

 - A note, either as an alphanumeric note name (e.g. cs4) or a numeric hertz
   frequency. Note that if one or more pitche objects have been initialised
   then it is not necessary to specify the octave as the last octave will be
   used

OPTIONAL ARGUMENTS:

keyword arguments:
- :src-ref-pitch. A note-name symbol indicating the perceived fundamental
  pitch of a given digital audio file, to allow for later transposition of
  that audio file using note-names. Default: C4
- :midi-channel. An integer indicating which MIDI channel is to be used for
  playback of this pitch. Default: 1

RETURN VALUE:

- A pitch object.

EXAMPLE:

```
;; Make a pitch object using a note-name symbol
(make-pitch 'c4)

=>
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
       pitch-bend: 0.0
       degree: 120, data-consistent: T, white-note: C4
       nearest-chromatic: C4
       src: 1.0, src-ref-pitch: C4, score-note: C4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
       show-accidental: T, white-degree: 28,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4

;; Make a pitch object using a frequency in hertz and including a value for the
;; keyword argument :midi-channel, then print the DATA and MIDI-NOTE slots to
;; see the method's automatic conversion for those values.
(let ((p (make-pitch 261.63 :midi-channel 1)))
  (print (data p))
  (print (midi-note p)))

=>
C4
60

;; Make a pitch object for use with a digital audio file that includes a
;; note-name symbol for the sample-rate-conversion reference pitch; then print
```

```
;; the SRC slot of the resulting pitch object
(let ((p (make-pitch 'c4 :src-ref-pitch 'a4)))
  (src p))

=> 0.5946035487490308

;;  make two pitch objects; the 2nd uses the octave of the first
(progn (make-pitch 'cs6)
       (make-pitch 'a))
=>
PITCH: frequency: 1760.000, midi-note: 93, midi-channel: 1
       pitch-bend: 0.0
       degree: 186, data-consistent: T, white-note: A6
       nearest-chromatic: A6
...
```

SYNOPSIS:

```
(defun make-pitch (pitch &key (src-ref-pitch 'c4) (midi-channel 1))
```

28.5.45   pitch/midi-

[pitch] [Methods]

DESCRIPTION:

Determine the difference in number of semitones between the values of the
MIDI values of two given pitch objects.

NB: This method does not return absolute difference; instead, it may return
    positive or negative results depending on the order in which the pitch
    objects are given. (This will aid in revealing directionality.)

ARGUMENTS:

 - A first pitch object.
 - A second pitch object.

RETURN VALUE:

 Returns a number. The number may be positive or negative.

EXAMPLE:

```
;; Subtracting the lower pitch object from the higher returns a positive number
```

```
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (midi- p1 p2))

=> 2

;; Reversing the order in which the pitch objects are entered may return a
;; negative number
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (midi- p2 p1))

=> -2
```

SYNOPSIS:

```
(defmethod midi- ((p1 pitch) (p2 pitch))
```

28.5.46   pitch/midi-play-pitch-list

[pitch] [Functions]

DATE:

 January 2nd 2020, Heidhausen

DESCRIPTION:

 Generate a MIDI file from a list of pitches.

ARGUMENTS:

 the pitch list: either pitch objects or symbols

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :rhythm. The rhythm (symbol or object) to be used for all pitches. Default
   = 'e (eighth note)
 - :tempo. The MIDI file tempo. Default = 120BPM.
 - :file. The patch of the MIDI file to write. Default = "/tmp/tmp.mid"
```

RETURN VALUE:

 the MIDI file path (string)

SYNOPSIS:

```
(defun midi-play-pitch-list (pitches
                              &key (rhythm 'e) (tempo 120) (file "/tmp/tmp.mid"))
```

28.5.47   pitch/no-accidental

[pitch] [Metadata]

DESCRIPTION:

 Set the SHOW-ACCIDENTAL and ACCIDENTAL-IN-PARENTHESES slots of a specified
 pitch object to NIL, preventing any accidentals or accidentals in
 parentheses from being shown for that event in the printed score.

 NB: This will only be effective if the :respell-notes option for
     cmn-display and write-lp-data-for-all is set to NIL.

ARGUMENTS:

 - A pitch object.

RETURN VALUE:

 Always NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c4 cs4 fs4))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((2 4) - s s s s - - s s s s -))
                                 :pitch-seq-palette ((1 2 3 2 1 2 3 2)))))
        :rthm-seq-map '((1 ((vn (1))))))))
  (no-accidental (pitch-or-chord (get-note mini 1 7 'vn)))
  (cmn-display mini :respell-notes nil)
  (write-lp-data-for-all mini :respell-notes nil))
```

SYNOPSIS:

```
(defmethod no-accidental ((p pitch))
```

28.5.48   pitch/note=

[pitch] [Methods]

DESCRIPTION:

 Tests to see the note-name symbols (values in the DATA slots) of two given
 pitch objects are equal.

 NB: This method allows for the comparison of pitch objects created using
     frequency numbers and those created using note-name symbols.

ARGUMENTS:

 - A first pitch object.
 - A second pitch object.

RETURN VALUE:

 T if the note-name symbols of the given pitch objects are equal, otherwise
 NIL.

EXAMPLE:

```
;; Two pitch objects with equal note-name symbols return T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (note= p1 p2))

=> T
;; Two pitch objects with unequal note-name symbols return F
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (note= p1 p2))

=> NIL

;; Pitch objects created using frequency numbers and those created using
;; note-name symbols can be effectively compared using this method
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 261.63)))
  (note= p1 p2))

=> T
```

SYNOPSIS:

```
(defmethod note= ((p1 pitch) (p2 pitch) &optional ignore)
```

28.5.49   pitch/pitch-

[pitch] [Methods]

DESCRIPTION:

Get the distance in semitones between the values of two pitch objects. This
method also takes fractional values into consideration. The

ARGUMENTS:

- A first pitch object.
- A second pitch object.

RETURN VALUE: EXAMPLE:

```
;; Get the distance between two "white-keys"
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
 (pitch- p1 p2))

=> 2.0

;; Get the distance in semitones between two frequencies (rounded to the
;; nearest degree, which by default is quarter-tones)
(let ((p1 (make-pitch 293.66))
      (p2 (make-pitch 261.63)))
 (pitch- p1 p2))

=> 2.0

;; Getting the distance in semitones between pitches with fractional values can
;; return fractional results
(let ((p1 (make-pitch 'dqs4))
      (p2 (make-pitch 'c4)))
 (pitch- p1 p2))

=> 2.5
```

SYNOPSIS:

```
(defmethod pitch- ((p1 pitch) (p2 pitch))
```

28.5.50   pitch/pitch-class-eq

[pitch] [Methods]

DATE:

 14 Aug 2010

DESCRIPTION:

 Test whether the values of two pitch objects are of the same pitch class,
 i.e. both Cs, or F#s,irrespective of octave.

ARGUMENTS:

 - A first pitch object.
 - A second pitch object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not enharmonic pitches are considered
   equal. T = enharmonic pitches are considered equal. Default = NIL.

RETURN VALUE:

 T if the values of the two pitch objects are of the same pitch class,
 otherwise NIL.

EXAMPLE:

```
;; A comparison of two pitch objects with values of the same pitch class
;;; returns T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c5)))
  (pitch-class-eq p1 p2))

=> T

;; A comparison of two pitch objects with values of differing pitch classes
;; returns NIL
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'cs5)))
  (pitch-class-eq p1 p2))

=> NIL

;; A comparison of two pitch objects with enharmonically equivalent pitch
;; classes returns NIL by default
(let ((p1 (make-pitch 'c4))
```

```
      (p2 (make-pitch 'bs4)))
  (pitch-class-eq p1 p2))

=> NIL

;; Setting the optional argument to T causes the method to consider
;; enharmonically equivalent pitch classes equal
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs4)))
  (pitch-class-eq p1 p2 t))

=> T
```

SYNOPSIS:

```
(defmethod pitch-class-eq ((p1 pitch) (p2 pitch)
                             &optional enharmonics-are-equal)
```

28.5.51   pitch/pitch-in-range

[pitch] [Methods]

DESCRIPTION:

 Determine whether the frequency of a given pitch object falls between the
 frequencies of two other given pitch objects.

ARGUMENTS:

 - A first pitch object.
 - A second pitch object, which must be lower than the third.
 - A third pitch object, which must be higher than the second.

RETURN VALUE:

 T if the frequency value of the first specified pitch object falls between
 the second and third specified pitch objects, otherwise NIL.

EXAMPLE:

```
;; The method returns T when the frequency value of the first pitch object
;; falls between that of the second and third pitch objects.
(let ((p (make-pitch 'c4))
      (l (make-pitch 'g3))
      (h (make-pitch 'a7)))
```

```
  (pitch-in-range p l h))

=> T

;; The method returns NIL when the frequency value of the first pitch object is
;; below the range designated by the frequency values of the other two objects.
(let ((p (make-pitch 'g3))
      (l (make-pitch 'c4))
      (h (make-pitch 'a7)))
  (pitch-in-range p l h))

=> NIL

;; The method returns NIL when the frequency value of the first pitch object is
;; above the range designated by the frequency values of the other two objects.
(let ((p (make-pitch 'a7))
      (l (make-pitch 'g3))
      (h (make-pitch 'c4)))
  (pitch-in-range p l h))

=> NIL

;; The method will also return NIL if the frequency value of the second pitch
;; object is higher than that of the third
(let ((p (make-pitch 'c4))
      (l (make-pitch 'a7))
      (h (make-pitch 'g3)))
  (pitch-in-range p l h))

=> NIL
```

SYNOPSIS:

```
(defmethod pitch-in-range ((p pitch) (lowest pitch) (highest pitch))
```

28.5.52   pitch/pitch-inc

[pitch] [Methods]

DESCRIPTION:

Increment the value of a given pitch object by one degree (default) or by a
specified number of degrees (optional argument).

NB: The slippery-chicken package uses a quarter-tone degree system by

default, so any function or method involving a degree argument will be
measured in quarter-tones, not semitones. Thus, while the MIDI note
value for 'C4 is 60 (chromatic semitones), (note-to-degree 'C4) will
return 120. Thus, this method will increment by one quarter-tone by
default, and any value the chooser uses for the optional argument is
also number of quarter-tones.

 NB: This method returns a new pitch object rather than modifying the values
    of the original.

ARGUMENTS:

 - A pitch object.

OPTIONAL ARGUMENTS:

 - A number indicating the step (in degrees) by which the pitch value is to
   be incremented. Defaults = 1.

RETURN VALUE:

 Returns a pitch object.

EXAMPLE:

```
;; The method by default returns a pitch object and increments by one
;; quarter-tone
(let ((p (make-pitch 'c4)))
  (pitch-inc p))

=>
PITCH: frequency: 269.292, midi-note: 60, midi-channel: 0
      pitch-bend: 0.5
      degree: 121, data-consistent: T, white-note: C4
      nearest-chromatic: C4
      src: 1.0293022394180298, src-ref-pitch: C4, score-note: CS4
      qtr-sharp: 1, qtr-flat: NIL, qtr-tone: 1,
      micro-tone: T,
      sharp: NIL, flat: NIL, natural: NIL,
      octave: 4, c5ths: 0, no-8ve: CQS, no-8ve-no-acc: C
      show-accidental: T, white-degree: 28,
      accidental: QS,
      accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: CQS4, tag: NIL,
```

```
data: CQS4
```

```
;; Using the optional argument, increment steps can be changed; for example,
;; here to one semitone (= 2 quarter-tones)
(let ((p (make-pitch 'c4)))
  (data (pitch-inc p 2)))
```

```
=> CS4
```

```
;; Here the method increments by 4 quarter-tones = 1 whole-tone
(let ((p (make-pitch 'c4)))
  (data (pitch-inc p 4)))
```

```
=> D4
```

```
;; Incrementing by an additional number of quarter-tones at each pass
(let ((p (make-pitch 'c4)))
  (loop for i from 0 to 4 collect (data (pitch-inc p i))))
```

```
=> (C4 CQS4 CS4 DQF4 D4)
```

SYNOPSIS:

```
(defmethod pitch-inc ((p pitch) &optional (degrees 1))
```

28.5.53   pitch/pitch-intersection

[pitch] [Functions]

DESCRIPTION:

 Return pitch objects whose values consist of pitches common to two given
 lists of pitch items. The given lists of pitch items can consist of pitch
 objects or note-name symbols, or one list of one type and the second of the
 other.

ARGUMENTS:

 - A first list of pitch objects.
 - A second list of pitch objects.

RETURN VALUE:

 Returns a list of pitch objects that are common to both original lists.

EXAMPLE:

```
;; Returns a list of pitch objects
(let ((p1 '(c4 d4 e4 f4))
      (p2 (loop for nn in '(d4 e4 f4 g4) collect (make-pitch nn))))
  (pitch-intersection p1 p2))

(
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
[...]
data: D4
[...]
PITCH: frequency: 329.628, midi-note: 64, midi-channel: 0
[...]
data: E4
[...]
PITCH: frequency: 349.228, midi-note: 65, midi-channel: 0
[...]
data: F4
[...]
)
```

SYNOPSIS:

```
(defun pitch-intersection (pitch-list1 pitch-list2)
```

28.5.54   pitch/pitch-list-to-symbols

[pitch] [Functions]

DESCRIPTION:

 Return as a list the note-name values from a given list of pitch objects.

ARGUMENTS:

 - A list of pitch objects.

OPTIONAL ARGUMENTS:

 - The package in which to process the list of pitches. Default = :sc.

RETURN VALUE:

 A list of note-name symbols

EXAMPLE:

```
;; Create a list of pitch objects and apply the pitch-list-to-symbols method
(let ((pl))
  (setf pl (loop for m from 0 to 127 by 13
              collect (make-pitch (midi-to-note m))))
  (pitch-list-to-symbols pl))

=> (C-1 CS0 D1 EF2 E3 F4 FS5 G6 AF7 A8)
```

SYNOPSIS:

```
(defun pitch-list-to-symbols (pitch-list &optional (package :sc))
```

28.5.55   pitch/pitch-max

[pitch] [Methods]

DESCRIPTION:

Determine which of two specified pitch objects has the greater frequency
value and return that pitch object.

NB: If the two frequency values are equal, the method returns a pitch
    object equivalent to both.

ARGUMENTS:

- A first pitch object.
- A second ptich object.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
;; Compare two pitch objects and return the one with the greater frequency
;; value
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch-max p1 p2))

=>
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
       pitch-bend: 0.0
       degree: 124, data-consistent: T, white-note: D4
```

```
        nearest-chromatic: D4
        src: 1.1224620342254639, src-ref-pitch: C4, score-note: D4
        qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
        micro-tone: NIL,
        sharp: NIL, flat: NIL, natural: T,
        octave: 4, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
        show-accidental: T, white-degree: 29,
        accidental: N,
        accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: D4, tag: NIL,
data: D4

;; Comparing two pitch objects with equal frequency values returns a pitch
;; object equal to both
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (data (pitch-max p1 p2)))

=> C4
```

SYNOPSIS:

```
(defmethod pitch-max ((p1 pitch) (p2 pitch))
```

28.5.56   pitch/pitch-member

[pitch] [Functions]

DESCRIPTION:

 Test whether a specified pitch is a member of a given list of pitches.

 This function can take pitch objects, note-name symbols or numerical
 frequency values (or lists thereof) as its arguments.

ARGUMENTS:

 - A pitch item. This may be a pitch object, a note-name symbol or a
   numerical frequency value.
 - A list of pitch items. These items may be pitch objects, note-name
   symbols, or numerical frequency values.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether or not the function should consider
  enharmonically equivalent pitches to be equal. T = enharmonics are
  considered equal. Default = T.
- The second optional argument allows the user to specify the test for
  comparison, such as note=, pitch-class-eq, or the default pitch=. If the
  user wants to specify his or her own, the test must take three arguments:
  p1, p2 and <enharmonics-are-equivalent> (which may of course be ignored).

RETURN VALUE:

Similar to Lisp's "member" function, this method returns the tail of the
tested list starting with the specified pitch if the pitch is indeed a
member of that list, otherwise returns NIL. NB: The list returned is a list
of pitch objects.

EXAMPLE:

```
;; Returns NIL if the specified pitch item is not a member of the given list
(let ((pl '(c4 d4 e4)))
  (pitch-member 'f4 pl))

=> NIL

;; Returns the tail of the given list starting from the specified pitch if that
;; pitch is indeed a member of the tested list
(let ((pl '(c4 d4 e4)))
  (pitch-list-to-symbols (pitch-member 'd4 pl)))

=> (D4 E4)

;; Enharmonically equivalent pitches are considered equal by default
(let ((pl '(c4 ds4 e4)))
  (pitch-list-to-symbols (pitch-member 'ef4 pl)))

=> (DS4 E4)

;; Enharmonic equivalence can be turned off by setting the first optional
;; argument to NIL
(let ((pl '(c4 ds4 e4)))
  (pitch-list-to-symbols (pitch-member 'ef4 pl nil)))

=> NIL
```

SYNOPSIS:

```
(defun pitch-member (pitch pitch-list
```

```
                        &optional (enharmonics-are-equal t)
                                  (test #'pitch=))
```

28.5.57   pitch/pitch-min

[pitch] [Methods]

DESCRIPTION:

 Determine which of two specified pitch objects has the lower frequency
 value and return that pitch object.

 NB: If the two frequency values are equal, the method returns a pitch
     object equivalent to both.

ARGUMENTS:

 - A first pitch object.
 - A second ptich object.

RETURN VALUE:

 A pitch object.

EXAMPLE:

```
;; Compare two pitch objects and return the one with the lower frequency value
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch-min p1 p2))

=>
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
       pitch-bend: 0.0
       degree: 120, data-consistent: T, white-note: C4
       nearest-chromatic: C4
       src: 1.0, src-ref-pitch: C4, score-note: C4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
       show-accidental: T, white-degree: 28,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
```

```
NAMED-OBJECT: id: C4, tag: NIL,
data: C4

;; Comparing two pitch objects with equal frequency values returns a pitch
;; object equal to both
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (data (pitch-min p1 p2)))

=> C4
```

SYNOPSIS:

```
(defmethod pitch-min ((p1 pitch) (p2 pitch))
```

28.5.58   pitch/pitch-or-chord=

[pitch] [Methods]

DATE:

 Tue 29 Oct 2019 08:57:53 GMT - Warwick

DESCRIPTION:

 Convenience method, test to see if the pitch-or-chord slots of two event
 objects are the same.

ARGUMENTS:

 - a pitch or chord object
 - a pitch or chord object

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not enharmonic pitches are considered
   equal. T = enharmonic pitches are considered equal. Default = NIL.
 - a number to indicate the frequency deviation allowed before returning NIL.

RETURN VALUE:

 T if the values of the two specified pitch or chord objects are equal, otherwise
 NIL.

EXAMPLE:

```
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (pitch-or-chord= p1 p2))

=> T

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2))

=> NIL

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2 t))

=> T

(let ((p1 (make-pitch 'c4))
      (c1 (make-chord '(c4 e4 g4))))
  (pitch-or-chord= p1 c1))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(c4 e4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs4 ff4 g4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs3 ff4 g4))))
  (pitch-or-chord= c1 c2 t))

=> T

;; Chords with only one pitch are the same as pitch object with the same pitch.
(let ((c (make-chord '(c4)))
      (p (make-pitch 'c4)))
```

```
  (pitch-or-chord= c p))
```

=> T

SYNOPSIS:

```
(defmethod pitch-or-chord= ((p1 pitch) (p2 pitch)
                               &optional enharmonics-are-equal
                                 (frequency-tolerance 0.01))
```

28.5.59   pitch/pitch-round

[pitch] [Methods]

DESCRIPTION:

 Rounds the value of a specified pitch object to the nearest chromatic
 semitone (non-microtonal MIDI) pitch.

ARGUMENTS:

 - A pitch object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :as-symbol. T or NIL to indicate whether the method is to return an
   entire pitch object or just a note-name symbol of the new pitch. NIL = a
   new pitch object. Default = NIL.
 - :package. Used to identify a separate Lisp package in which to process
   result. This is really only applicable is combination with :as-symbol set
   to T. Default = :sc.

RETURN VALUE:

 A pitch object by default.

 If the :as-symbol argument is set to T, then a note-name symbol is returned
 instead.

EXAMPLE:

```
;; Returns a pitch object by default; here an example rounding a quarter-tone
;;; note-name symbol to the nearest chromatic pitch
```

```
(let ((p (make-pitch 'CQS4)))
  (pitch-round p))

=>
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
NAMED-OBJECT: id: C4, tag: NIL,
data: C4

;; Also rounds frequencies to the nearest chromatic pitch. This example first
;; prints the original values automatically stored with frequency 269.0
;; (rounded by default to the nearest quarter-tone), then the new value rounded
;; to the nearest chromatic semitone
(let ((p (make-pitch 269.0)))
  (print (data p))
  (print (pitch-round p :as-symbol t)))

=>
CQS4
C4
```

SYNOPSIS:

```
(defmethod pitch-round ((p pitch) &key (as-symbol nil) (package :sc))
```

28.5.60   pitch/pitch<

[pitch] [Methods]

DESCRIPTION:

Test to see if the frequency value of one specified pitch object is less
than that of a second.

NB: Due to the fact that a given note-name may encompass several
    fractionally different frequencies (e.g. both 261.626 and 261.627 are
    both considered to be C4), this method is not suitable for comparing
    pitch objects of which one was created using a frequency and the other
    was created using a note-name symbol.

ARGUMENTS:

- A pitch object.
- A second pitch object.

RETURN VALUE:

 Returns T if the frequency value of the first pitch object is less than
 that of the second, otherwise NIL.

EXAMPLE:

```
;; T is returned when the frequency of the first pitch is less than that of
;; the second
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch< p1 p2))

=> T

;; NIL is returned when the frequency of the first pitch is not less than
;; that of the second
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch< p2 p1))

=> NIL

;; Equivalent pitches return NIL
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (pitch< p2 p1))

=> NIL

;; This method can be effectively used to compare the frequency values of two
;; pitch objects that were both created using frequency numbers
(let ((p1 (make-pitch 261.63))
      (p2 (make-pitch 293.66)))
  (pitch< p1 p2))

=> T

;; Due to sc's numerical accuracy, this method is not suitable for comparing
;; pitch objects of which one was created using a note-name symbol and the
;; other was created using a numerical frequency value. Such comparisons may
;; return misleading results.
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 261.63)))
  (pitch< p1 p2))

=> T
```

SYNOPSIS:

```
(defmethod pitch< ((p1 pitch) (p2 pitch))
```

28.5.61   pitch/pitch<=

[pitch] [Methods]

DESCRIPTION:

Test to see if the frequency value of one specified pitch object is less
than or equal to than that of a second.

NB: Due to the fact that a given note-name may encompass several
    fractionally different frequencies (e.g. both 261.626 and 261.627 are
    both considered to be C4), this method is not suitable for comparing
    pitch objects of which one was created using a frequency and the other
    was created using a note-name symbol.

ARGUMENTS:

- A pitch object.
- A second pitch object.

RETURN VALUE:

Returns T if the frequency value of the first pitch object is less than or
equal to that of the second, otherwise NIL.

EXAMPLE:

```
;; T is returned when the frequency of the first pitch is less than or equal to
;; that of the second
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch<= p1 p2))

=> T

;; NIL is returned when the frequency of the first pitch is not less than or
;; equal to that of the second
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'd4)))
  (pitch<= p2 p1))
```

```
=> NIL

;; Equivalent pitches return T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (pitch<= p2 p1))

=> T

;; This method can be effectively used to compare the frequency values of two
;; pitch objects that were both created using frequency numbers
(let ((p1 (make-pitch 261.63))
      (p2 (make-pitch 293.66)))
  (pitch<= p1 p2))

=> T

;; Due to sc's numerical accuracy, this method is not suitable for comparing
;; pitch objects of which one was created using a note-name symbol and the
;; other was created using a numerical frequency value. Such comparisons may
;; return misleading results.
(let ((p1 (make-pitch 261.63))
      (p2 (make-pitch 'c4)))
  (pitch<= p1 p2))

=> NIL
```

SYNOPSIS:

```
(defmethod pitch<= ((p1 pitch) (p2 pitch))
```

28.5.62   pitch/pitch=

[pitch] [Methods]

DESCRIPTION:

Determines if the note-name and chromatic semitone MIDI values of two
specified pitch objects are the same (or very close to each other in the
case of frequency and src slot comparison).

By default, this method returns NIL when comparing enharmonic pitches. This
can behavior can be changed by setting the optional argument to T, upon
which enharmonic pitches are considered equal.

NB: This method may return NIL when comparing pitch objects created using
    frequencies with those created using note-names. The method
    pitch::note= may be more useful in this case.

NB: Pitch objects created using frequencies are only considered equal if
    their frequency values are within 0.01Hz of each other.

ARGUMENTS:

 - A first pitch object.
 - A second pitch object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not enharmonic pitches are considered
   equal. T = enharmonic pitches are considered equal. Default = NIL.
 - a number to indicate the frequency deviation allowed before returning NIL.

RETURN VALUE:

 T if the values of the two specified pitch objects are equal, otherwise
 NIL.

EXAMPLE:

```
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 'C4)))
  (pitch= p1 p2))

=> T

;; Comparison of unequal pitch objects created using note-name symbols returns
NIL
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 'D4)))
  (pitch= p1 p2))

=> NIL

;; Comparison of enharmonically equivalent pitch objects returns NIL by default
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'CS4))
      (p2 (make-pitch 'DF4)))
  (pitch= p1 p2))
```

```
=> NIL

;; Comparison of enharmonically equivalent pitch objects return T when the
;; optional argument is set to T
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 'C4)))
  (pitch= p1 p2 t))

=> T

;; Comparison of pitch objects created using frequencies with those created
;; using note-name symbols returns T
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 261.63)))
  (pitch= p1 p2))

=> T
```

SYNOPSIS:

```
(defmethod pitch= ((p1 pitch) (p2 pitch) &optional enharmonics-are-equal
                    (frequency-tolerance 0.01)) ; (src-tolerance 0.0001))
```

28.5.63   pitch/pitch>

[pitch] [Methods]

DESCRIPTION:

 Test to see if the frequency value of one specified pitch object is greater
 than that of a second.

 NB: Due to the fact that a given note-name may encompass several
 fractionally different frequencies (e.g. both 261.626 and 261.627 are both
 considered to be C4), this method is not suitable for comparing pitch
 objects of which one was created using a frequency and the other was
 created using a note-name symbol.

ARGUMENTS:

 - A pitch object.
 - A second pitch object.

RETURN VALUE:

 Returns T if the frequency value of the first pitch object is greater than
 that of the second, otherwise NIL.

EXAMPLE:

```
;; T is returned when the frequency of the first pitch is greater than that of
;; the second
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (pitch> p1 p2))

=> T

;; NIL is returned when the frequency of the first pitch is not greater than
;; that of the second
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (pitch> p2 p1))

=> NIL

;; Equivalent pitches return NIL
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'd4)))
  (pitch> p2 p1))

=> NIL

;; This method can be effectively used to compare the frequency values of two
;; pitch objects that were both created using frequency numbers
(let ((p1 (make-pitch 293.66))
      (p2  (make-pitch 261.63)))
  (pitch> p1 p2))

=> T

;; Due to sc's numerical accuracy, this method is not suitable for comparing
;; pitch objects of which one was created using a note-name symbol and the
;; other was created using a numerical frequency value. Such comparisons may
;; return misleading results.
(let ((p1 (make-pitch 261.63))
      (p2 (make-pitch 'c4)))
  (pitch> p1 p2))

=> T
```

SYNOPSIS:

(defmethod pitch> ((p1 pitch) (p2 pitch))

28.5.64   pitch/pitch>=

[pitch] [Methods]

DESCRIPTION:

 Test to see if the frequency value of one specified pitch object is greater
 than or equal to than that of a second.

 NB: Due to the fact that a given note-name may encompass several
     fractionally different frequencies (e.g. both 261.626 and 261.627 are
     both considered to be C4), this method is not suitable for comparing
     pitch objects of which one was created using a frequency and the other
     was created using a note-name symbol.

ARGUMENTS:

 - A pitch object.
 - A second pitch object.

RETURN VALUE:

 Returns T if the frequency value of the first pitch object is greater than
 or equal to that of the second, otherwise NIL.

EXAMPLE:

```
;; T is returned when the frequency of the first pitch is greater than or equal
;;; to that of the second
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (pitch>= p1 p2))

=> T

;; NIL is returned when the frequency of the first pitch is not greater than or
;; equal to that of the second
(let ((p1 (make-pitch 'd4))
      (p2 (make-pitch 'c4)))
  (pitch>= p2 p1))
```

```
=> NIL

;; Equivalent pitches return T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (pitch>= p2 p1))

=> T

;; This method can be effectively used to compare the frequency values of two
;; pitch objects that were both created using frequency numbers
(let ((p1 (make-pitch 293.66))
      (p2 (make-pitch 261.63)))
  (pitch>= p1 p2))

=> T

;; Due to sc's numerical accuracy, this method is not suitable for comparing
;; pitch objects of which one was created using a note-name symbol and the
;; other was created using a numerical frequency value. Such comparisons may
;; return misleading results.
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 261.63)))
  (pitch>= p1 p2))

=> NIL
```

SYNOPSIS:

```
(defmethod pitch>= ((p1 pitch) (p2 pitch))
```

28.5.65   pitch/print-simple-pitch-list

[pitch] [Functions]

DESCRIPTION:

 Print the data symbols of a list of pitch objects. NB If you need to print
 these in the context of another printing routine, you can get a string by
 passing NIL as the stream.

DATE:

 April 10th 2012

ARGUMENTS:

- A simple list of pitch objects.

OPTIONAL ARGUMENTS:

- The stream to print to (e.g. an open file). Default: the Lisp Terminal
  (REPL).

RETURN VALUE:

 The list of pitch data symbols.

EXAMPLE:

```
(print-simple-pitch-list (init-pitch-list '(c4 d4 e4)))
=>
(C4 D4 E4)
(C4 D4 E4)
```

SYNOPSIS:

```
(defun print-simple-pitch-list (pitch-list &optional (stream t))
```

28.5.66   pitch/remove-octaves

[pitch] [Functions]

DESCRIPTION:

 Removes all but one of any pitch items in a given list that have the same
 pitch-class but different octaves, keeping the lowest instance only.

 The list of pitch items may be a list of pitch objects or a list of
 note-name symbols.

ARGUMENTS:

- A list of pitch items. These may be pitch objects or note-name symbols.

OPTIONAL ARGUMENTS:

 keyword arguments
- :as-symbol. T or NIL indicating whether the object is to return a list of
  pitch objects or a list of note-name symbols. T = return pitch
  objects. Default = NIL.

- :package. Used to identify a separate Lisp package in which to itern
  result. This is really only applicable is combination with :as-symbol set
  to T. Default = :sc.
- :allow. T or NIL to indicate whether pitch objects of certain, specified
  octave-doublings are to be kept even if they are not the lowest. This
  argument takes the form of either a single number or a list of
  numbers. NB: This number does not indicate the octave in which the pitch
  object is found, but rather pitch objects that are the specified number
  of octaves above the lowest instance of the pitch class. Thus, :allow 2
  indicates keeping the lowest pitch plus any instances of the same pitch
  class two octaves above that lowest pitch (i.e.,
  double-octaves). However, it is important to note that the function first
  removes any octave doublings that are not excepted by the :allow
  argument, which may produce confusing results. Given a list of
  consecutive octaves, such as '(C1 C2 C3 C4) and an :allow value of 2, the
  function will first remove any equal pitch classes that are are not 2
  octaves apart, resulting in C2, C3, and C4 being removed as they are one
  octave distant from C1, C2 and C3. The result of the function using these
  values would therefore be '(C1).

RETURN VALUE:

 Returns a list of pitch objects by default. If the keyword argument
 :as-symbol is set to T, the method returns a list of note-name symbols
 instead.

 If the first element of the pitch list is a number (i.e. a frequency), the
 method returns a list of frequencies.

EXAMPLE:

```
;; The method returns a list of pitch objects by default
(remove-octaves '(c1 c2 c3 g3))

=> (
PITCH: frequency: 32.703, midi-note: 24, midi-channel: 0
[...]
data: C1
[...]
PITCH: frequency: 195.998, midi-note: 55, midi-channel: 0
[...]
data: G3
[...]
)

;; If the first element of the pitch list is a frequency, the method returns a
```

```
;; list of frequencies
(remove-octaves '(261.63 523.26 1046.52 196.00))
```

```
=> (261.63 196.0)
```

```
;; Setting keyword argument :as-symbol to T returns a list of note-name symbols
;; instead
(remove-octaves '(261.63 523.26 1046.52 196.00) :as-symbol t)
```

```
=> (C4 G3)
```

SYNOPSIS:

```
(defun remove-octaves (pitch-list &key as-symbol allow (package :sc))
```

28.5.67 pitch/remove-pitches

[pitch] [Functions]

DESCRIPTION:

Remove a list of specified pitch items from a given list of pitch
items. Even if only one pitch item is to be removed it should be stated as
a list.

The pitch items can be in the form of pitch objects, note-name symbols or
numerical frequency values.

ARGUMENTS:

- A list of pitch items from which the specified list of pitches is to be
  removed. These can take the form of pitch objects, note-name symbols or
  numerical frequency values.
- A list of pitch items to remove from the given list. These can take the
  form of pitch objects, note-name symbols or numerical frequency
  values. Even if only one pitch is to be removed is must be stated as a
  list.

OPTIONAL ARGUMENTS:

keyword arguments:
- :enharmonics-are-equal. Set to T or NIL to indicate whether or not
  enharmonically equivalent pitches are to be considered the same pitch. T
  = enharmonically equivalent pitches are equal. Default = T.
- :return-symbols. Set to T or NIL to indicate whether the function is to

    return a list of pitch objects or note-name symbols. T = note-name
    symbols. Default = NIL.

RETURN VALUE:

 Returns a list of pitch objects by default. When the keyword argument
 :return-symbols is set to T, the function will return a list of note-names
 instead.

 If the specified list of pitches to be removed are not found in the given
 list, the entire list is returned.

EXAMPLE:

```
;; By default the function returns a list of pitch objects
(let ((pl '(c4 d4 e4)))
  (remove-pitches pl '(d4 e4)))

=> (
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
[...]
)

;; Setting the keyword argument :return-symbols to T causes the function to
;; return a list of note-name symbols instead. Note in this example too that
;; even when only one pitch item is being removed, it must be stated as a list.
(let ((pl '(261.62 293.66 329.62)))
  (remove-pitches pl '(293.66) :return-symbols t))

=> (C4 E4)

;; The function will also accept pitch objects
(let ((pl (loop for n in '(c4 d4 e4) collect (make-pitch n))))
  (remove-pitches pl '(,(make-pitch 'e4)) :return-symbols t))

=> (C4 D4)

;; By default the function considers enharmonically equivalent pitches to be
;; equal
(let ((pl (loop for n in '(c4 ds4 e4) collect (make-pitch n))))
  (remove-pitches pl '(ef4) :return-symbols t))

=> (C4 E4)
```

```
;; This feature can be turned off by setting the :enharmonics-are-equal keyword
;; argument to NIL. In this case here, the specified pitch is therefore not
;; found in the given list and the entire original list is returned.
(let ((pl (loop for n in '(c4 ds4 e4) collect (make-pitch n))))
  (remove-pitches pl '(ef4)
                  :return-symbols t
                  :enharmonics-are-equal nil))

=> (C4 DS4 E4)
```

SYNOPSIS:

```
(defun remove-pitches (pitch-list remove
                       &key (enharmonics-are-equal t)
                            (return-symbols nil))
```

28.5.68   pitch/round-to-nearest

[pitch] [Methods]

DATE:

 January 30th 2017

DESCRIPTION:

 Round the pitch object to the nearest pitch in the current or given scale,
 i.e. force frequency, midi-note, pitch bend, etc. to conform to this nearest
 pitch.

 NB rounding to the nearest using a microtonal :scale argument might cause
 errors if the global (in-scale ...) scale is :chromatic because although
 chromatic-scale is a subset of quarter-tone and twelfth-tone (so all
 chromatic-pitches work) if you have microtonal pitches generated e.g. by
 frequency arguments to make-pitch the note symbol in the microtonal scale
 doesn't exist in :chromatic. E.g. the following breaks:

 (in-scale :chromatic)
 264Hz is approx. c 1/12 sharp
 (round-to-nearest (make-pitch 264) :scale 'twelfth-tone)
 -> fails because rounding in 1/12 tone scale gives us CTS4, which doesn't
 exist in the global chromatic scale. Of course all is well if we do this
 beforehand:
 (in-scale :twelfth-tone)

ARGUMENTS:

 the pitch object

OPTIONAL ARGUMENTS:

 keyword argument:
 :scale. The scale to use when rounding. (Common Music tuning object or
 symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or 'quarter-tone
 only at present. Default is the current scale as set by (in-scale :...).

RETURN VALUE:

 the modified pitch object

EXAMPLE:

```
(make-pitch 443)
-->
PITCH: frequency: 443.000, midi-note: 69, midi-channel: 1
       pitch-bend: 0.12
       degree: 138, data-consistent: T, white-note: A4
       nearest-chromatic: A4
       src: 1.6932597032572d0, src-ref-pitch: C4, score-note: A4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: T,
...

(round-to-nearest (make-pitch 443))
-->
PITCH: frequency: 440.000, midi-note: 69, midi-channel: 1
       pitch-bend: 0.0
       degree: 138, data-consistent: T, white-note: A4
       nearest-chromatic: A4
       src: 1.6817929, src-ref-pitch: C4, score-note: A4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
...

;;; exact quarter tones round down, not up (true for quarter-sharps and flats):
(round-to-nearest (make-pitch 'dqs7) :scale 'chromatic-scale)
-->
PITCH: frequency: 2349.318, midi-note: 98, midi-channel: 1
       pitch-bend: 0.0
       degree: 196, data-consistent: T, white-note: D7
       nearest-chromatic: D7
       src: 8.979696, src-ref-pitch: C4, score-note: D7
```

```
        qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
        micro-tone: NIL,
        sharp: NIL, flat: NIL, natural: T,
        octave: 7, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
        show-accidental: T, white-degree: 57,
        accidental: N,
        accidental-in-parentheses: NIL, marks: NIL,
        marks-before: NIL
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: D7, tag: NIL,
data: D7
**************
```

SYNOPSIS:

```
(defmethod round-to-nearest ((p pitch) &key (scale cm::*scale*))
```

28.5.69 pitch/set-midi-channel

[pitch] [Methods]

DESCRIPTION:

Set the MIDI-CHANNEL slot of the given pitch object.

The method takes two mandatory arguments in addition to the given pitch
object, the first being the MIDI-channel used for non-microtonal pitch
objects, the second that used for microtonal pitch objects.

NB: The pitch object only has one MIDI-CHANNEL slot, and determines whether
    that slot is set to the specified non-microtonal or microtonal
    midi-channel argument based on whether or not the pitch of the given
    pitch object is determined to be a microtone or not.

ARGUMENTS:

- A pitch object.
- A number indicating the MIDI channel which is to be used to play back
  non-microtonal pitches.

OPTIONAL ARGUMENTS:

- A number indicating the MIDI channel which is to be used to play back

microtonal pitches. NB: See player.lsp/make-player for details on
microtones in MIDI output. Default = NIL which means it will take on the
value of the second argument (all pitches on the same channel, whether
microtonal or not).

RETURN VALUE:

A number indicating which value has been set to the given pitch object's
MIDI-CHANEL slot.

EXAMPLE:

```
;; When the pitch of the given pitch object is non-microtonal, the method sets
;; that pitch object's MIDI-CHANNEL slot to the first value specified.
(let ((p (make-pitch 'c4)))
  (set-midi-channel p 11 12)
  (midi-channel p))

=> 11

;; When the pitch of the given pitch object is microtonal, the method sets
;; that pitch object's MIDI-CHANNEL slot to the second value specified.
(let ((p (make-pitch 'cqs4)))
  (set-midi-channel p 11 12))

=> 12
```

SYNOPSIS:

```
(defmethod set-midi-channel ((p pitch) midi-channel
                                &optional microtones-midi-channel)
```

28.5.70   pitch/sort-pitch-list

[pitch] [Functions]

DESCRIPTION:

Sort a list of pitch objects from low to high based on their frequency
value.

ARGUMENTS:

- A list of pitch objects.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the method is to return a list of pitch
  objects or a list of note-name symbols.
- The package in which the operation is to be performed. Default = :sc.

RETURN VALUE:

 Returns a list of pitch objects by default. When the first optional
 argument is set to T, the method returns a list of note-name symbols
 instead.

EXAMPLE:

```
;; Create a list of pitch objects by passing downward through a series of MIDI
;; values and print the result. Then apply the sort-pitch-list method and print
;; the result of that to see the list now ordered from low to high.
(let ((pl))
  (setf pl (loop for m from 64 downto 60
              collect (make-pitch (midi-to-note m))))
  (print (loop for p in pl collect (data p)))
  (print (sort-pitch-list pl)))

=>
(E4 EF4 D4 CS4 C4)
(
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
[...]
PITCH: frequency: 277.183, midi-note: 61, midi-channel: 0
[...]
data: CS4
[...]
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
[...]
data: D4
[...]
PITCH: frequency: 311.127, midi-note: 63, midi-channel: 0
[...]
data: EF4
[...]
PITCH: frequency: 329.628, midi-note: 64, midi-channel: 0
[...]
data: E4
```

```
)

;; Setting the first optional argument to T causes the method to return a list
;; of note-name symbols instead
(let ((pl))
  (setf pl (loop for m from 64 downto 60
              collect (make-pitch (midi-to-note m))))
  (sort-pitch-list pl t))

=> (C4 CS4 D4 EF4 E4)
```

SYNOPSIS:

```
(defun sort-pitch-list (pitch-list &optional
                           (return-symbols nil)
                           (package :sc))
```

28.5.71   pitch/transpose

[pitch] [Methods]

DESCRIPTION:

Transpose the pitch information (frequency, note-name, midi-note etc.) of a
given pitch object by a specified number of semitones. The number of
semitones specified can be fractional; however, all fractional values will
be rounded to the nearest quarter-tone frequency (or default smallest step
for the current scale).

NB: This method returns a new pitch object rather than altering the values
    of the current pitch object.

ARGUMENTS:

 - A pitch object.
 - A number representing the number of semitones to be transposed, and which
   can be fractional.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :as-symbol. T or NIL to indicate whether the method is to return an
   entire pitch object or just a note-name symbol of the new pitch. NIL = a
   new pitch object. Default = NIL.
 - :package. Used to identify a separate Lisp package in which to process

the result. This is really only applicable is combination with :as-symbol
set to T. Default = :sc.

RETURN VALUE:

 A pitch object by default.

 If the :as-symbol argument is set to T, then a note-name symbol is returned
 instead.

EXAMPLE:

```
;; By default the method returns a pitch object
(let ((p (make-pitch 'c4)))
  (transpose p 2))

=>
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
       pitch-bend: 0.0
       degree: 124, data-consistent: T, white-note: D4
       nearest-chromatic: D4
       src: 1.1224620342254639, src-ref-pitch: C4, score-note: D4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 4, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
       show-accidental: T, white-degree: 29,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: D4, tag: NIL,
data: D4

;; Setting the :as-symbol keyword argument to T returns just the note-name
;; symbol of the new pitch instead
(let ((p (make-pitch 'c4)))
  (transpose p 2 :as-symbol t))

=> D4

;; The semitones argument can be set to a decimal-point fraction, which may
;; result in quarter-tone pitch values being returned
(let ((p (make-pitch 'c4)))
  (transpose p 2.5))

=>
```

```
PITCH: frequency: 302.270, midi-note: 62, midi-channel: 0
       pitch-bend: 0.5
       degree: 125, data-consistent: T, white-note: D4
       nearest-chromatic: D4
       src: 1.1553527116775513, src-ref-pitch: C4, score-note: DS4
       qtr-sharp: 1, qtr-flat: NIL, qtr-tone: 1,
       micro-tone: T,
       sharp: NIL, flat: NIL, natural: NIL,
       octave: 4, c5ths: 0, no-8ve: DQS, no-8ve-no-acc: D
       show-accidental: T, white-degree: 29,
       accidental: QS,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: DQS4, tag: NIL,
data: DQS4


;; Fractional semitone arguments are automatically rounded to the nearest
;; quarter-tone, causing x.5 and x.7, for example, to return the same result,
;; while x.3 and x.1 will return the same value as the given integer
(let ((p (make-pitch 'c4)))
  (print (transpose p 2 :as-symbol t))
  (print (loop for s from 0 to 4
            collect (transpose p (+ 2 (* s .1)) :as-symbol t)))
  (print (loop for s from 5 to 9
            collect (transpose p (+ 2 (* s .1)) :as-symbol t))))

=>
D4
(D4 D4 D4 D4 D4)
(DQS4 DQS4 DQS4 DQS4 DQS4)
```

SYNOPSIS:

```
(defmethod transpose ((p pitch) semitones &key (as-symbol nil) (package :sc)
                       ignore)
```

28.5.72   pitch/transpose-pitch-list

[pitch] [Functions]

DESCRIPTION:

Transpose the values of a list of pitch objects by a specified number of
semitones.

ARGUMENTS:

 - A list of pitch objects.
 - A number indicating the number of semitones by which the list is to be
   transposed.

OPTIONAL ARGUMENTS:

 keyword arguments
 - :return-symbols. T or NIL indicating whether the method is to return a
   list of pitch objects or a list of note-name symbols for those pitch
   objects. T = note-name symbols. Default = NIL.
 - :package. The name of the package to perform the transpositions. Default =
   :sc.
 - :lowest. Don't transpose pitches which are lower than this
   argument. Default = C-1 (midi note 0)
 - :lowest. Don't transpose pitches which are higher than this
   argument. Default = B8 (midi note 119)

RETURN VALUE:

 By default, the method returns a list of pitch objects. When the first
 optional argument is set to T, a list of note-name symbols is returned
 instead.

EXAMPLE:

```
;; Create a list of pitch objects and apply the transpose-pitch-list method
;; with the semitones argument set to 2
(let ((pl))
  (setf pl (loop for m from 60 to 71 collect (make-pitch (midi-to-note m))))
  (transpose-pitch-list pl 2))

=>
(
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
[...]
PITCH: frequency: 311.127, midi-note: 63, midi-channel: 0
[...]
PITCH: frequency: 329.628, midi-note: 64, midi-channel: 0
[...]
PITCH: frequency: 349.228, midi-note: 65, midi-channel: 0
[...]
PITCH: frequency: 369.994, midi-note: 66, midi-channel: 0
[...]
PITCH: frequency: 391.995, midi-note: 67, midi-channel: 0
[...]
```

```
PITCH: frequency: 415.305, midi-note: 68, midi-channel: 0
[...]
PITCH: frequency: 440.000, midi-note: 69, midi-channel: 0
[...]
PITCH: frequency: 466.164, midi-note: 70, midi-channel: 0
[...]
PITCH: frequency: 493.883, midi-note: 71, midi-channel: 0
[...]
PITCH: frequency: 523.251, midi-note: 72, midi-channel: 0
[...]
PITCH: frequency: 554.365, midi-note: 73, midi-channel: 0
[...]
)


;; Perform the same action with the return-symbols optional argument set to T
(let ((pl))
  (setf pl (loop for m from 60 to 71 collect (make-pitch (midi-to-note m))))
  (print (transpose-pitch-list pl 2 t)))

=> (D4 EF4 E4 F4 FS4 G4 AF4 A4 BF4 B4 C5 CS5)
```

SYNOPSIS:

```
(defun transpose-pitch-list (pitch-list semitones
                             &key
                                 (return-symbols nil)
                                 (package :sc)
                                 lowest highest)
```

28.5.73   pitch/transpose-pitch-list-to-octave

[pitch] [Functions]

DESCRIPTION:

 Transpose the pitch values of a list of pitch objects into a specified
 octave. The individual initial pitch objects can have initial pitch values
 of different octaves.

ARGUMENTS:

 - A list of pitch objects.
 - A number indicating the octave in which the resulting list should be.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :as-symbols. Set to T or NIL to indicate whether the method is to return
  a list of pitch objects or a list of the note-name symbols from those
  pitch objects. T = return as symbols. Default = NIL.
- :package. Used to identify a separate Lisp package in which to itern
  result. This is really only applicable is combination with :as-symbol set
  to T. Default = :sc.
- :remove-duplicates. Set to T or NIL to indicate whether any duplicate
  pitch objects are to be removed from the resulting list.  T = remove
  duplicates. Default = T.
```

RETURN VALUE:

```
Returns a list of pitch objects by default. When the keyword argument
:as-symbols is set to T, the method returns a list of note-name symbols
instead.
```

EXAMPLE:

```
;; Create a list of four pitch objects from random MIDI numbers and print it,
;; then apply transpose-pitch-list-to-octave, setting the octave argument to 4,
;; and print the result
(let ((pl))
  (setf pl (loop repeat 4 collect (make-pitch (midi-to-note (random 128)))))
  (print (loop for p in pl collect (data p)))
  (print (transpose-pitch-list-to-octave pl 4)))

=>
(CS7 F7 B0 D4)
(
PITCH: frequency: 493.883, midi-note: 71, midi-channel: 0
[...]
data: B4
[...]
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
[...]
data: D4
[...]
PITCH: frequency: 277.183, midi-note: 61, midi-channel: 0
[...]
data: CS4
[...]
PITCH: frequency: 349.228, midi-note: 65, midi-channel: 0
[...]
data: F4
)
```

```
;; Setting the keyword argument :as-symbols to T return a list of note-names
;; instead
(let ((pl))
  (setf pl (loop repeat 4 collect (make-pitch (midi-to-note (random 128)))))
  (print (loop for p in pl collect (data p)))
  (print (transpose-pitch-list-to-octave pl 4 :as-symbols t)))

=>
(D5 E1 C7 AF1)
(E4 AF4 D4 C4)

;; The method removes duplicate pitch objects from the resulting list by
;; default
(let ((pl))
  (setf pl (loop repeat 4 collect (make-pitch (midi-to-note (random 128)))))
  (print (loop for p in pl collect (data p)))
  (print (transpose-pitch-list-to-octave pl 4 :as-symbols t)))

=>
(B7 AF1 AF7 G1)
(G4 AF4 B4)
```

SYNOPSIS:

```
(defun transpose-pitch-list-to-octave (pitch-list octave
                                       &key
                                       as-symbols
                                       (package :sc)
                                       (remove-duplicates t))
```

28.5.74   pitch/transpose-to-octave

[pitch] [Methods]

DESCRIPTION:

Transpose the values of a given pitch object to a specified octave.

NB: This method creates a new pitch object rather than replacing the values
of the original.

ARGUMENTS:

- A pitch object.
- A number indicating the new octave.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :as-symbol. T or NIL to indicate whether the method is to return an
  entire pitch object or just a note-name symbol of the new pitch. NIL = a
  new pitch object. Default = NIL.
- :package. Used to identify a separate Lisp package in which to process
  the result. This is really only applicable is combination with :as-symbol
  set to T. Default = :sc.
```

RETURN VALUE:

```
A pitch object by default.

If the :as-symbol argument is set to T, then a note-name symbol is returned
instead.
```

EXAMPLE:

```
;; Transpose the values of a pitch object containing middle-C (octave 4) to the
;;; C of the treble clef (octave 5)
(let ((p (make-pitch 'c4)))
  (transpose-to-octave p 5))

=>
PITCH: frequency: 523.251, midi-note: 72, midi-channel: 0
       pitch-bend: 0.0
       degree: 144, data-consistent: T, white-note: C5
       nearest-chromatic: C5
       src: 2.0, src-ref-pitch: C4, score-note: C5
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 5, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
       show-accidental: T, white-degree: 35,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C5, tag: NIL,
data: C5

;; Setting the :as-symbol argument to T returns a note-name symbol instead of a
;; pitch object
(let ((p (make-pitch 'c4)))
  (transpose-to-octave p 5 :as-symbol t))
```

```
=> C5
```

SYNOPSIS:

```
(defmethod transpose-to-octave ((p pitch) new-octave
                                &key
                                (as-symbol nil)
                                (package :sc))
```

28.5.75   linked-named-object/player

[linked-named-object] [Classes]

NAME:

```
 player

 File:             player.lsp

 Class Hierarchy:  named-object -> linked-named-object -> player

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of the player class which holds an
                   instrument or a assoc-list of instruments in it's data
                   slot.

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    7th September 2001

 $$ Last modified:  09:31:33 Thu Jan 10 2019 CET

 SVN ID: $Id$
```

28.5.76   player/make-player

[player] [Functions]

DESCRIPTION:

 Create a player object from a specified instrument-palette object and a

specified instrument or list of instruments which that player plays.

The player object is separate from the instrument object as on player in an
ensemble may perform more than one instrument ("double"), such as flute and
piccolo, clarinet and bass clarinet, or sax, flute and clarinet.

ARGUMENTS:

- A symbol which will be the ID of the resulting player object.
- An instrument-palette object. If NIL then
  +slippery-chicken-standard-instrument-palette+ will be used
- A symbol or a list of symbols that are the instruments from the
  specified instrument-palette object that the given player will play, as
  spelled and defined within the instrument-palette object. NB: If only one
  instrument is to be assigned to the given player, it should be stated as
  symbol rather than a list, to avoid errors in the DOUBLES slot.

OPTIONAL ARGUMENTS:

keyword arguments:
- :midi-channel. An integer that indicates the MIDI channel on which any
  non-microtonal pitch material for this player is to be played
  back. Default = 1.
- :microtones-midi-channel. An integer that indicates the MIDI channel on
  which any microtonal pitch material for this player is to be played
  back. slippery chicken uses this channel to add MIDI pitch-bends via CM
  so that microtonal chords are possible, but due to a current glitch these
  tracks contain no pitch-bend data. A work-around for this is to simply
  open the MIDI file in a sequencer and shift the entire channel by the
  desired pitch-bend value. Default = -1.
- :cmn-staff-args. A list of pairs that indicate any additional arguments
  to the call to cmn::staff for this player, such as staff size, number of
  lines etc. Instead of being real cmn function calls, as they would be in
  normal cmn, this is a simple list of pairs; e.g. '(staff-size .8
  staff-lines 3). Defaults = NIL.
- :staff-names. A symbol, string or list of staff names, generally strings,
  for each instrument the player will play. E.g. '("violin II"). If not
  given, then the instrument's name as defined in the instrument palette
  will be used. Default = NIL.
- :staff-short-names. A symbol, string or list of short staff
  names. E.g. '("vln2"). Default = NIL

RETURN VALUE:

Returns a player object.

EXAMPLE:

```
;; Create a player object with just one instrument object
(let ((ip (make-instrument-palette
            'inst-pal
            '((picc (:transposition-semitones 12 :lowest-written d4
                        :highest-written c6))
              (flute (:lowest-written c4 :highest-written d7))
              (clar (:transposition-semitones -2 :lowest-written e3
                        :highest-written c6))
              (horn (:transposition f :transposition-semitones -7
                        :lowest-written f2 :highest-written c5))
              (vln (:lowest-written g3 :highest-written c7 :chords t))
              (vla (:lowest-written c3 :highest-written f6 :chords t))))))
  (make-player 'player-one ip 'flute))

=>
PLAYER: (id instrument-palette): INST-PAL
doubles: NIL, cmn-staff-args: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: PLAYER-ONE, tag: NIL,
data:
INSTRUMENT: lowest-written:
[...]
NAMED-OBJECT: id: FLUTE, tag: NIL,
data: NIL

;; Create a player object with two instruments, setting the midi channels using
;; the keyword arguments, then print the corresponding slots to see the changes
(let* ((ip (make-instrument-palette
            'inst-pal
            '((picc (:transposition-semitones 12 :lowest-written d4
                        :highest-written c6))
              (flute (:lowest-written c4 :highest-written d7))
              (clar (:transposition-semitones -2 :lowest-written e3
                        :highest-written c6))
              (horn (:transposition f :transposition-semitones -7
                        :lowest-written f2 :highest-written c5))
              (vln (:lowest-written g3 :highest-written c7 :chords t))
              (vla (:lowest-written c3 :highest-written f6 :chords t)))))
       (plr (make-player 'player-one ip '(flute picc)
                          :midi-channel 1
                          :microtones-midi-channel 2)))
  (print (loop for i in (data (data plr)) collect (id i)))
  (print (midi-channel plr))
  (print (microtones-midi-channel plr)))
```

```
=>
(FLUTE PICC)
1
2

;;; With specified cmn-staff-args
(let ((ip (make-instrument-palette
            'inst-pal
            '((picc (:transposition-semitones 12 :lowest-written d4
                      :highest-written c6))
              (flute (:lowest-written c4 :highest-written d7))
              (clar (:transposition-semitones -2 :lowest-written e3
                      :highest-written c6))
              (horn (:transposition f :transposition-semitones -7
                      :lowest-written f2 :highest-written c5))
              (vln (:lowest-written g3 :highest-written c7 :chords t))
              (vla (:lowest-written c3 :highest-written f6 :chords t))))))
  (make-player 'player-one ip '(flute picc)
               :midi-channel 1
               :microtones-midi-channel 2
               :cmn-staff-args '(staff-size .8 staff-lines 3)))

=>
PLAYER: (id instrument-palette): INST-PAL
doubles: T, cmn-staff-args: (#<SELF-ACTING {10097B6E73}>
                            #<SELF-ACTING {10097B6EE3}>), total-notes: 0, total-degrees: 0,
total-duration: 0.000, total-bars: 0, tessitura: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: PLAYER-ONE, tag: NIL,
data:
[...]
```

SYNOPSIS:

```
(defun make-player (id instrument-palette instruments
                    &key (cmn-staff-args nil) staff-names staff-short-names
                        (microtones-midi-channel -1) (midi-channel 1))
```

28.5.77   player/microtonal-chords-p

[player] [Methods]

DESCRIPTION:

 Determines whether the MICROTONES-MIDI-CHANNEL slot of the given player

object is set to a value greater than 0 and is different to the midi-channel
slot, indicating that the player and its instrument are capable of
performing microtonal chords.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 Returns T if the value stored in the MICROTONES-MIDI-CHANNEL slot of the
 given player object is greater than 0, otherwise returns NIL.

EXAMPLE:

```
;; Returns T
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'vln ip 'violin :microtones-midi-channel 2)))
  (microtonal-chords-p plr))

=> T

;; Returns NIL
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'pno ip 'piano)))
  (microtonal-chords-p plr))

=> NIL
```

SYNOPSIS:

```
(defmethod microtonal-chords-p ((p player))
  ;; MDE Thu Dec 28 17:49:36 2017 -- updated as now make-player sets microtones
  ;; channel to midi-channel if it's not explicitly set
  (and (integer>0 (microtones-midi-channel p))
       (/= (microtones-midi-channel p)
           (midi-channel p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod score-write-bar-line ((p player))
  (let* ((data (data p))
         (ins (if (typep data 'assoc-list)
                  (first (data data))
                data)))
    (score-write-bar-line ins)))
```

28.5.78   player/player-get-instrument

[player] [Methods]

DESCRIPTION:

Get the instrument object assigned to a single-instrument player object or
get the specified instrument object assigned to a multiple-instrument
player object.

NB: This method will drop into the debugger with an error if no optional
    argument is supplied when applying the method to a multiple-instrument
    player object. It will also print a warning when supplying an optional
    argument to a player object that contains only one instrument object.

ARGUMENTS:

 - A player object.

OPTIONAL ARGUMENTS:

 - Actually a required object for multiple-instrument player objects: The
   symbol that is the ID of the sought-after instrument object, as it
   appears in the instrument-palette with which the player object which
   made. If the given player object consists of only one instrument object,
   this argument is disregarded and a warning is printed. If this argument
   is simply T then the first instrument is returned (in the assoc-list, not
   the piece, of which we have no knowledge).

RETURN VALUE:

 Returns an instrument object.

EXAMPLE:

```
;; Returns an instrument object. Needs no optional argument when applied to a
;; player object that contains only one instrument object
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'pno ip 'piano)))
   (player-get-instrument plr))

=>
INSTRUMENT:
[...]
NAMED-OBJECT: id: PIANO, tag: NIL,
```

```
data: NIL

;; Returns the only existing instrument object and prints a warning if using
;; the optional argument when applying to a single-instrument player object
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'pno ip 'piano)))
  (id (player-get-instrument plr 'piano)))

=> PIANO
WARNING:
   player::player-get-instrument: player PNO has only 1 instrument so optional
argument PIANO is being ignored

;; Asking for a non-existent instrument obect from a single-instrument player
;; object returns the only existing instrument object instead
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'pno ip 'piano)))
  (id (player-get-instrument plr 'marimba)))

=> PIANO
WARNING:
   player::player-get-instrument: player PNO has only 1 instrument so optional
argument PIANO is being ignored

;; The ID desired instrument object must be specified when applying the method
;; to a multiple-instrument player object
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'percussion ip '(marimba vibraphone))))
  (id (player-get-instrument plr 'marimba)))

=> MARIMBA

;; Interrupts and drops into the debugger when the optional argument is omitted
;; in applying the method to a multiple-instrument player object
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'percussion ip '(marimba vibraphone))))
  (player-get-instrument plr))

=>
player::player-get-instrument: PERCUSSION doubles so you need to pass the ID of
the instrument you want.
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod player-get-instrument ((p player) &optional ins (warn t))
```

28.5.79   player/plays-transposing-instrument

[player] [Methods]

DESCRIPTION:

 Determine whether a given player object has one or more transposing
 instrument objects assigned to it.

ARGUMENTS:

 - A player object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether instruments that transpose at the octave are
   to be considered transposing instruments. T = instruments that transpose
   at the octave are not considered transposing instruments. Default = T.

RETURN VALUE:

 Returns T if one or more of the instrument objects assigned to the given
 player object has a transposition value other than C or a
 transposition-semitones value other than 0.

EXAMPLE:

```
;; Create a player object using the 'b-flat-clarinet instrument object
;; definition from the default +slippery-chicken-standard-instrument-palette+,
;; then apply the method.
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'cl ip 'b-flat-clarinet)))
  (plays-transposing-instrument plr))

=> T

;; Create a player object using the 'flute instrument object definition from
;; the default +slippery-chicken-standard-instrument-palette+, then apply the
;; method.
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'fl ip 'flute)))
  (plays-transposing-instrument plr))

=> NIL
```

```
;; Although the intended procedure is to list single instruments as once-off
;; symbols (as in the previous example), single instruments can also be added
;; as a one-item list
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'fl ip '(flute))))
  (doubles plr))

=> NIL

;; Create a player object using a list that consists of the 'flute and
;; 'alto-sax instrument object definitions from the default
;; +slippery-chicken-standard-instrument-palette+, then apply the method to see
;; that it returns T even when only one of the instruments is transposing.
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'fl ip '(flute alto-sax))))
  (plays-transposing-instrument plr))

=> T

;; Setting the optional argument to NIL causes instruments that transpose at
;; the octave to return T.
(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'db ip 'double-bass)))
  (plays-transposing-instrument plr))

=> NIL

(let* ((ip +slippery-chicken-standard-instrument-palette+)
       (plr (make-player 'db ip 'double-bass)))
  (plays-transposing-instrument plr nil))

=> T
```

SYNOPSIS:

```
(defmethod plays-transposing-instrument ((p player)
                                         &optional (ignore-octaves t) ignore)
```

28.5.80   player/reset-instrument-stats

[player] [Methods]

DATE:

 23rd August 2013

DESCRIPTION:

 Reset the statistics slots for each instrument the player plays.

ARGUMENTS:

 - The player object

OPTIONAL ARGUMENTS:

 - just-total-duration. If NIL update all statistics slots, otherwise just
   the total-duration slot of each instrument

RETURN VALUE:

 T if the player has instruments, NIL if not.

SYNOPSIS:

(defmethod reset-instrument-stats ((p player) &optional just-total-duration)

28.5.81   player/tessitura-degree

[player] [Methods]

DESCRIPTION:

 Return a number that represents the average pitch for a specified
 instrument over the course of a piece. The number returned will be degrees
 in the current scale.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 A number that is the tessitura-degree; i.e., average pitch of the given
 instrument for the entirety of the given musical data.

EXAMPLE:

(let ((mini
      (make-slippery-chicken

```
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                    (va (violin :midi-channel 2))
                    (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (va (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (tessitura-degree (get-data 'vc (ensemble mini))))
```

=> 136

SYNOPSIS:

```
(defmethod tessitura-degree ((p player))
```

28.5.82   player/tessitura-note

[player] [Methods]

DESCRIPTION:

Return the value of the TESSITURA-DEGREE slot of a specified player object
as a note-name symbol.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 - A note-name symbol.

EXAMPLE:

```
(in-scale :chromatic)

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (violin :midi-channel 2))
```

```
                        (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (va (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (tessitura-note (first (data (ensemble mini)))))
```

=> BF3

SYNOPSIS:

```
(defmethod tessitura-note ((p player))
```

28.5.83   player/total-bars

[player] [Methods]

DESCRIPTION:

 Return the number of bars in a specified player object.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 - An integer.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (violin :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
```

```
                                    (va (1 1 1 1 1))
                                    (vc (1 1 1 1 1)))))))))
  (total-bars (first (data (ensemble mini)))))
```

=> 5

SYNOPSIS:

```
(defmethod total-bars ((p player))
```

28.5.84   player/total-degrees

[player] [Methods]

DESCRIPTION:

Return a number that reflects the mean note (tessitura) of a player's
part. This is calculated by incrementing the TOTAL-DEGREES slot of the
corresponding instrument object for each attacked note in the player's part
by the degree of that note (in the scale of the piece), and then dividing
the sum by the total number of notes in the player's part.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 - An integer.

EXAMPLE:

```
(in-scale :chromatic)

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                    (va (violin :midi-channel 2))
                    (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
```

```
                                     (va (1 1 1 1 1))
                                     (vc (1 1 1 1 1)))))))))
  (total-degrees (first (data (ensemble mini)))))
```

=> 865

SYNOPSIS:

```
(defmethod total-degrees ((p player))
```

28.5.85   player/total-duration

[player] [Methods]

DESCRIPTION:

 Get the total duration of played notes for a given player over the span of
 a piece.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 A number that is the total duration in seconds of played notes.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (violin :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3))))
                            (2 ((((2 4) (q) e (s) 32 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (va (2 2 2 2 2))
                            (vc (1 2 1 2 1)))))))))
  (print (total-duration (get-data 'vn (ensemble mini)))))
```

```
  (print (total-duration (get-data 'va (ensemble mini))))
  (print (total-duration (get-data 'vc (ensemble mini)))))
```

```
=>
6.875
3.75
5.625
```

SYNOPSIS:

```
(defmethod total-duration ((p player))
```

28.5.86   player/total-notes

[player] [Methods]

DESCRIPTION:

 Get the total number of notes (actually events) played by a specified
 player (not rests or tied notes, but midi-notes) in the piece which this
 instrument plays. A chord counts as 1 note/event.

ARGUMENTS:

 - A player object.

RETURN VALUE:

 - An integer that is the number of notes for that player.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (va (violin :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((gs3 as3 b3 cs4 ds4 e4 fs4 gs4 as4 b4 cs5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (va (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
```

```
  (print (total-notes (get-data 'vc (ensemble mini)))))
```

```
=> 15
```

SYNOPSIS:

```
(defmethod total-notes ((p player))
```

28.5.87   linked-named-object/rhythm

[linked-named-object] [Classes]

NAME:

```
 rhythm

 File:              rhythm.lsp

 Class Hierarchy:  named-object -> linked-named-object -> rhythm

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of the rhythm class for parsing and
                   storing the properties of rhythms.

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    11th February 2001

 $$ Last modified:  16:29:10 Thu Oct  8 2020 CEST

 SVN ID: $Id$
```

28.5.88   rhythm/accented-p

[rhythm] [Methods]

DATE:

```
 05 Apr 2011
```

DESCRIPTION:

Check the MARKS slot of a given rhythm object to determine if it possesses
an accent mark. The rhythm object may also possess other marks as well.

ARGUMENTS:

 - A rhythm object.

RETURN VALUE:

 If the accent mark ('a) is indeed found in the MARKS slot of the given
 rhythm object, the tail of the list of marks contained in that slot is
 returned; otherwise NIL is returned.

EXAMPLE:

```
;; Make a rhythm object, add an accent, and test for the presence of the accent
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (accented-p r))

=> (A)

;; Check if an accent mark is among all marks in the MARKS slot
(let ((r (make-rhythm 'q)))
  (add-mark-once r 's)
  (add-mark-once r 'a)
  (accented-p r))

=> (A S)

;; Add an accent and staccato, then remove the accent and test for it
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (add-mark-once r 's)
  (rm-marks r 'a)
  (accented-p r))

=> NIL
```

SYNOPSIS:

```
(defmethod accented-p ((r rhythm))
```

28.5.89   rhythm/add

[rhythm] [Methods]

DESCRIPTION:

Create a new rhythm object with a duration that is equal to the sum of the
duration of two other given rhythm objects.

NB: This method only returns a single rhythm rather than a list with
    ties. Thus q+s, for example, returns TQ...

If the resulting duration cannot be presented as a single, notatable
rhythm, the DATA slot of the resulting rhythm object is set to NIL, though
the VALUE and DURATION slots are still set with the corresponding numeric
values.

ARGUMENTS:

- A first rhythm object.
- A second rhythm object.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether a warning is printed when a rhythm cannot be
  made because the resulting value is 0 or a negative duration. Default =
  NIL (no warning issued).

RETURN VALUE:

A rhythm object. Returns NIL when the object cannot be made.

EXAMPLE:

```
;; A quarter plus an eighth makes a dotted quarter
(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'e)))
  (add r1 r2))

=>
RHYTHM: value: 2.6666666666666665, duration: 1.5, rq: 3/2, is-rest: NIL, score-rthm: 4.0f0.,
        undotted-value: 4, num-flags: 0, num-dots: 1, is-tied-to: NIL,
        is-tied-from: NIL, compound-duration: 1.5, is-grace-note: NIL,
        needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
        rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 4,
        tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q., tag: NIL,
data: Q.
```

```
;; A quarter plus a triplet-eighth is presented as a triplet-half
(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'te)))
  (data (add r1 r2)))

=> TH

;; A quarter plus a septuplet-16th cannot be represented as a single, notatable
;; rhythm and therefore produces an object with a VALUE and DURATION but no
;; DATA
(let ((r1 (make-rhythm 4))
      (r2 (make-rhythm 28)))
  (print (value (add r1 r2)))
  (print (duration (add r1 r2)))
  (print (data (add r1 r2))))

=>
3.5
1.1428571428571428
NIL
```

SYNOPSIS:

```
(defmethod add ((r1 rhythm) (r2 rhythm) &optional warn)
```

28.5.90   rhythm/add-mark

[rhythm] [Methods]

DESCRIPTION:

Add an articulation, dynamic, slur or any other mark to a rhythm (also
useful in the event subclass for changing note heads etc.) Multiple marks
can be added separately and consecutively to the same rhythm object.

A warning is printed if the same mark is added to the same rhythm object
more than once.

NB: This method checks to see if the mark added is a valid mark and will
    warn if it doesn't exist (but it will still add it, in case you have
    your own processing logic for it).

ARGUMENTS:

 - A rhythm object.

```
  - A mark.
```

OPTIONAL ARGUMENTS:

```
 - T or NIL to indicated whether to issue a warning when trying to add marks
   to a rest. Default = NIL.
```

RETURN VALUE:

```
 Always T.
```

EXAMPLE:

```
(let ((r (make-rhythm 'q)))
  (marks r))

=> NIL

(let ((r (make-rhythm 'q)))
  (add-mark r 'a))

=> T

(let ((r (make-rhythm 'q)))
  (add-mark r 's)
  (marks r))

=> (S)

(let ((r (make-rhythm 'q)))
  (add-mark r 'col-legno)
  (add-mark r 'as)
  (add-mark r 'x-head)
  (marks r))

=> (X-HEAD AS COL-LEGNO)

(let ((r (make-rhythm 'q)))
  (add-mark r 's)
  (add-mark r 's))

=> T
WARNING: rhythm::add-mark: S already present but adding again!:

(let ((r (make-rhythm 'e :is-rest t)))
```

```
  (add-mark r 'at)
  (print (is-rest r))
  (print (marks r)))

=>
T
(AT)

(let ((r (make-rhythm 'e :is-rest t)))
  (add-mark r 'at t))

=> T
WARNING:
[...]
rhythm::add-mark: add AT to rest?
```

SYNOPSIS:

```
(defmethod add-mark ((r rhythm) mark &optional warn-rest warn-again)
```

28.5.91   rhythm/add-mark-once

[rhythm] [Methods]

DATE:

 26 Jul 2011 (Pula)

DESCRIPTION:

 Apply the given mark to the given rhythm object, but do so only if the
 given rhythm object does not yet have the mark.

ARGUMENTS:

 - A rhythm object.
 - A mark.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not to print a warning when attempting to
   apply a mark to a rest.

RETURN VALUE:

Returns T if the mark is successfully applied (if the rhythm object did not
already possess the mark), otherwise NIL if the mark was not applied
because the rhythm object already had it.

EXAMPLE:

```
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a))

=> T

(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (marks r))

=> (A)

(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (add-mark-once r 'a))

=> NIL

(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (add-mark-once r 'a)
  (marks r))

=> (A)
```

SYNOPSIS:

```
(defmethod add-mark-once ((r rhythm) mark &optional warn-rest)
```

28.5.92   rhythm/begin-slur-p

[rhythm] [Methods]

DESCRIPTION:

Check to see if the MARKS slot of a given rhythm object contains a mark for
the beginning of a slur ('beg-sl). The rhythm object may also possess other
marks as well.

ARGUMENTS:

 - A rhythm object.

RETURN VALUE:

 If the 'beg-sl mark is indeed found in the MARKS slot of the given rhythm
 object, the tail of the list of marks contained in that slot is returned;
 otherwise NIL is returned.

EXAMPLE:

```
;; Create a rhythm object, add a 'beg-sl mark and check for it
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'beg-sl)
  (begin-slur-p r))

=> (BEG-SL)

;; Add several marks to a rhythm object and check for 'beg-sl
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s beg-sl) do (add-mark-once r m))
  (begin-slur-p r))

=> (BEG-SL S A)

;; Add a 'beg-sl mark to a rhythm object, then delete it and check for it
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'beg-sl)
  (rm-marks r 'beg-sl)
  (begin-slur-p r))

=> NIL
```

SYNOPSIS:

```
(defmethod begin-slur-p ((r rhythm))
```

28.5.93   rhythm/delete-beam

[rhythm] [Methods]

DESCRIPTION:

 Removes indication for the start (1) or end (0) of a beam from the BEAM
 slot of a given rhythm object, replacing them with NIL.

ARGUMENTS:

- A rhythm object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Manually set the beam of a rhythm object and delete it to see result NIL
(let ((r (make-rhythm 'e)))
  (setf (beam r) 1)
  (delete-beam r))

=> NIL

;; Make a rthm-seq-bar object with beam indications, then check the BEAM slot
;; of each rhythm object in the rthm-seq-bar object.
(let ((rsb (make-rthm-seq-bar '((2 4) - s s e - q))))
  (loop for r in (rhythms rsb) collect (beam r)))

=> (1 NIL 0 NIL)

;; Make a rthm-seq-bar object with beam indications, delete them all, then
;; check the beam slot of each rhythm object in the rthm-seq-bar object.
(let ((rsb (make-rthm-seq-bar '((2 4) - s s e - q))))
  (loop for r in (rhythms rsb) do (delete-beam r))
  (loop for r in (rhythms rsb) collect (beam r)))

=> (NIL NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod delete-beam ((r rhythm))
```

28.5.94   rhythm/delete-marks

[rhythm] [Methods]

DESCRIPTION:

 Delete any marks in the MARKS slot of an event object created within a
 rhythm object, replacing the entire list of the MARKS slot with NIL.

ARGUMENTS:

 - A rhythm object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; The method returns NIL
(let ((r (make-rhythm (make-event 'c4 'q))))
  (loop for m in '(a s pizz) do (add-mark-once r m))
  (delete-marks r))

=> NIL

;; Create a rhythm object consisting of an event object and print the default
;; contents of the MARKS slot. Set the MARKS slot to contain three marks and
;; print the result. Apply the delete-marks method and print the result.
(let ((r (make-rhythm (make-event 'c4 'q))))
  (print (marks r))
  (loop for m in '(a s pizz) do (add-mark-once r m))
  (print (marks r))
  (delete-marks r)
  (print (marks r)))

=>
NIL
(PIZZ S A)
NI
```

SYNOPSIS:

```
(defmethod delete-marks ((r rhythm))
```

28.5.95   rhythm/duration-secs

[rhythm] [Methods]

DESCRIPTION:

 Determine the absolute duration in seconds of a given rhythm object at a
 given quarter-note tempo. If no tempo is specified, a tempo of 60 is
 assumed.

ARGUMENTS:

 - A rhythm object.

OPTIONAL ARGUMENTS:

 - A numerical tempo value based on quarter-note beats per minute.

RETURN VALUE:

 A real number (floating point) representing the absolute duration of the
 given rhythm object in seconds.

EXAMPLE:

```
;; Determine the duration in seconds of a quarter note with a default tempo of
;;; quarter = 60
(let ((r (make-rhythm 'q)))
  (duration-secs r))

=> 1.0

;; Specifying a different tempo results in a different duration in seconds
(let ((r (make-rhythm 'q)))
  (duration-secs r 96))

=> 0.625
```

SYNOPSIS:

```
(defmethod duration-secs ((r rhythm) &optional (tempo 60))
```

28.5.96   rhythm/end-slur-p

[rhythm] [Methods]

DESCRIPTION:

 Check to see if the MARKS slot of a given rhythm object contains a mark for
 the ending of a slur ('end-sl). The rhythm object may also possess other
 marks as well.

ARGUMENTS:

 - A rhythm object.

RETURN VALUE:

If the 'end-sl mark is indeed found in the MARKS slot of the given rhythm
object, the tail of the list of marks contained in that slot is returned;
otherwise NIL is returned.

EXAMPLE:

```
;; Create a rhythm object, add a 'end-sl mark and check for it
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'end-sl)
  (end-slur-p r))

=> (END-SL)

;; Add several marks to a rhythm object and check for 'end-sl
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s end-sl) do (add-mark-once r m))
  (end-slur-p r))

=> (END-SL S A)

;; Add an 'end-sl mark to a rhythm object, then delete it and check for it
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'end-sl)
  (rm-marks r 'end-sl)
  (end-slur-p r))

=> NIL
```

SYNOPSIS:

```
(defmethod end-slur-p ((r rhythm))
```

28.5.97   rhythm/end-tie

[rhythm] [Methods]

DATE:

September 14th 2018, Heidhausen

DESCRIPTION:

return T or NIL depending on whether the rhythm/event is the end of a tie
(tied-to but not tied-from)

ARGUMENTS:

the rhythm/event object

RETURN VALUE:

 T or NIL

SYNOPSIS:

(defmethod end-tie ((r rhythm))

28.5.98   rhythm/event

[rhythm] [Classes]

NAME:

 event

 File: event.lsp

 Class Hierarchy: named-object -> linked-named-object -> rhythm -> event

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the event class which holds data for
                  the construction of an audible event, be it a midi note,
                  a sample (with corresponding sampling-rate conversion
                  factor) or chord of these types.

                  It is generally assumed that event instances will be
                  created from (copies of) rhythm instances by promotion
                  through the sc-change-class function, hence this class is
                  derived from rhythm.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: March 19th 2001

 $$ Last modified: 15:55:32 Tue Oct 27 2020 CET

 SVN ID: Id

28.5.99 event/add-arrow

[event] [Methods]

DATE:

 25 Jun 2011

DESCRIPTION:

 Adds a start-arrow mark to the given event object and stores text that is
 to be attached to the start and end of the given arrow for LilyPond
 output. This is a little more complex than the usual mark adding process,
 hence this separate method and it not being possible to add arrows to
 rthm-seq objects. Not available for CMN. See also the slippery-chicken
 class method add-arrow-to-events

 NB: A separate end-arrow mark should be attached to the note where the end
     text is to appear. Use end-arrow for this or (add-mark e 'end-arrow).

ARGUMENTS:

 - An event object.
 - A start-text string.
 - An end-text string.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not to print a warning when trying to
   attach an arrow and accompanying marks to a rest. Default = NIL.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;; Create an event object and see that the MARKS-BEFORE and MARKS slots are set
;;; to NIL by default
(let ((e (make-event 'c4 'q)))
  (print (marks-before e))
  (print (marks e)))

=>
NIL
```

```
NIL

;; Create an event object, apply the add-arrow method, and print the
;; corresponding slots to see the changes.
(let ((e (make-event 'c4 'q)))
  (add-arrow e "start here" "end here")
  (print (marks-before e))
  (print (marks e)))

=>
((ARROW "start here" "end here"))
(START-ARROW)

;; Create an event object that is a rest and apply the add-arrow method with
;; the optional argument set to T to see the warning printed.
(let ((e (make-event nil 'q)))
  (add-arrow e "start here" "end here" t))

=> T
event::add-arrow: add arrow to rest?
```

SYNOPSIS:

```
(defmethod add-arrow ((e event) start-text end-text &optional warn-rest)
```

28.5.100   event/add-clef

[event] [Methods]

DESCRIPTION:

Add a clef indication to the MARKS-BEFORE slot of the given event object.

NB: This method does not check that the clef-name added is indeed a clef,
nor does it check to see if other clefs have already been attached to the
same event object.

ARGUMENTS:

 - An event object.
 - A clef name (symbol).

OPTIONAL ARGUMENTS:

 - (Internal "ignore" arguments only; not needed by the user).

RETURN VALUE:

Returns the contents (list) of the MARKS-BEFORE slot if successful.

Returns NIL if the clef name is already present in the MARKS-BEFORE slot
and is therefore not added.

EXAMPLE:

```
;; Successfully adding a clef returns the contents of the MARKS-BEFORE slot
(let ((e (make-event 'c4 'q)))
  (add-clef e 'treble))

=> ((CLEF TREBLE))

;; Returns NIL if the clef name is already present
(let ((e (make-event 'c4 'q)))
  (add-clef e 'treble)
  (add-clef e 'treble))

=> NIL

;; Add a clef name to the marks-before slot and check that it's there
(let ((e (make-event 'c4 'q)))
  (add-clef e 'bass)
  (marks-before e))

=> ((CLEF BASS))
```

SYNOPSIS:

```
(defmethod add-clef ((e event) clef &optional (delete-others t) ignore1 ignore2)
```

28.5.101   event/add-mark

[event] [Methods]

DATE:

March 15th 2011

DESCRIPTION:

An :after method which in addition to the rhythm class's functionality will:
a) update the event's amplitude slot if we set a dynamic as a mark (note

that the :before method will similarly remove dynamics so we don't get
multiple dynamics attached to a single pitch);
b) as of 26/4/16 attach midi controller messages for pedal up/down if you
attach the marks ped, ped-up, or ped^, also una corda / tre corde

ARGUMENTS:

 - the event object
 - the mark (symbol)

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether amplitudes should be set if dynamic marks are
 set. Default = T.

RETURN VALUE: SYNOPSIS:

```
(defmethod add-mark :after ((e event) mark
                             &optional (update-amplitude t) warn-again)
```

28.5.102   event/add-midi-program-changes

[event] [Methods]

DATE:

 8th October 2020, Heidhausen

DESCRIPTION:

 Add programme change data to an event.

ARGUMENTS:

 - the event object to add to
 - the MIDI channel to add the program change to. This can either by a single
   integer or a list of (usually 2) integers (for microtonal channels)
 - the program change. If a number, just use this directly. If a symbol, get
   the program change for the instrument associated with this ID in the given
   instrument-palette

OPTIONAL ARGUMENTS:

 - the instrument-palette object to get the program change numbers from for
   the given instrument if a symbol is passed as the 3rd argument. Default:
   +slippery-chicken-standard-instrument-palette+

RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod add-midi-program-changes
    ((e event) chan pc
     &optional (ins-palette +slippery-chicken-standard-instrument-palette+))
```

28.5.103   event/add-pitches

[event] [Methods]

DESCRIPTION:

Add pitches to a non-rest event. This works whether the event is a single
pitch or a chord.

NB This adds to the sounding pitches, not written pitches
of transposing instruments. Events attached to transposing instruments will
not yet work with this method.

If pitch objects are given, then the midi-channel of the pitch is presumed
to be correct before this method is called. Symbols will default to MIDI
channel 1.

ARGUMENTS:

- The event object
- &rest: an arbitrary number of pitches: either pitch objects or symbols.

RETURN VALUE:

The same event but with the new pitches added.

EXAMPLE:

```
(let ((e1 (make-event 'c4 'q))
      (e2 (make-event '(c4 e4 g4) 'e)))
  (add-pitches e1 'cs3 'd5)
  (add-pitches e2 'cs2)))
```

SYNOPSIS:

```
(defmethod add-pitches ((e event) &rest pitches)
```

28.5.104   event/add-trill

[event] [Methods]

DATE:

 24 Sep 2011

DESCRIPTION:

 Used for adding pitched trills to printed score output. Adds the necessary
 values to the MARKS and MARKS-BEFORE slots of a given event object.

 NB: The main interface for adding trills by hand is
     slippery-chicken::trill, which is the class-method combination that
     should be accessed for this purpose.

 NB: This method will check to see if the specified trill marks are already
     present in the MARKS and MARKS-BEFORE slots. If they are, the method
     will print a warning but will add the specified trill marks anyway.

ARGUMENTS:

 - An event object.
 - A pitch-symbol for the trill note.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not to print a warning when attaching
   trill information to a rest. Default = NIL.

RETURN VALUE:

 Always returns T.

 NB: At the moment the method will also print the reminder warning that this
 is a LilyPond-only mark.

EXAMPLE:

```
;; Create an event object and print the contents of the MARKS-BEFORE and MARKS
;; slots to see that they're empty by default.
(let ((e (make-event 'c4 'q)))
  (print (marks-before e))
  (print (marks e)))
```

```
=>
NIL
NIL

;; Create an event object, add a trill to the note 'D4, and print the
;; corresponding slots to see the changes
(let ((e (make-event 'c4 'q)))
  (add-trill e 'd4)
  (print (marks-before e))
  (print (marks e)))

=>
WARNING:
rhythm::validate-mark: no CMN mark for BEG-TRILL-A (but adding anyway).

(BEG-TRILL-A)
((TRILL-NOTE D4))

;; By default the method adds prints no warning when adding a mark to a rest
;; (though it still prints the warning that there is no CMN mark)
(let ((e (make-event nil 'q)))
  (add-trill e 'd4)
  (print (marks-before e))
  (print (marks e)))

=>
WARNING:
rhythm::validate-mark: no CMN mark for BEG-TRILL-A (but adding anyway).

(BEG-TRILL-A)
((TRILL-NOTE D4))

;; Set the optional argument to T to have the method print a warning when
;; attaching a mark to a rest
(let ((e (make-event nil 'q)))
  (add-trill e 'd4 t)
  (print (marks-before e))
  (print (marks e)))

=>
event::add-trill: add trill to rest?
WARNING:
rhythm::validate-mark: no CMN mark for BEG-TRILL-A (but adding anyway).

(BEG-TRILL-A)
```

```
((TRILL-NOTE D4))

;; Adding a trill that is already there will result in a warning being printed
;; but will add the mark anyway
(let ((e (make-event 'c4 'q)))
  (loop repeat 4 do (add-trill e 'd4))
  (print (marks-before e))
  (print (marks e)))

=>
WARNING:
   rhythm::add-mark: (TRILL-NOTE D4) already present but adding again!:
[...]
(BEG-TRILL-A BEG-TRILL-A BEG-TRILL-A BEG-TRILL-A)
((TRILL-NOTE D4) (TRILL-NOTE D4) (TRILL-NOTE D4) (TRILL-NOTE D4))
```

SYNOPSIS:

```
(defmethod add-trill ((e event) trill-note &optional warn-rest)
```

28.5.105   event/common-notes

[event] [Methods]

DESCRIPTION:

Return the number of pitches common to two events. Note that the chord
method returns more information, including the common pitches and pitch
symbols.

ARGUMENTS:

- first event object
- second event object

OPTIONAL ARGUMENTS:

- T or NIL to treat enharmonic equivalents as equal
- T or NIL to treat the same pitches in different octaves as equal

RETURN VALUE:

the number of common pitches (integer). 2nd (and 3rd) returned values will
be the list of common pitch(es), also as a 3rd value: a list of symbols if
both events are chords).

SYNOPSIS:

```
(defmethod common-notes ((e1 event) (e2 event)
                         &optional (enharmonics-are-equal t)
                           (octaves-are-true nil))
```

28.5.106   event/delete-clefs

[event] [Methods]

DESCRIPTION:

Delete any clef names found in the MARKS-BEFORE slot of a given event
object.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not to print a warning when there are no
   clef marks to delete.
 - (Other internal "ignore" arguments only; not needed by the user).

RETURN VALUE:

 Always NIL.

EXAMPLE:

```
;; Returns NIL when no clef marks are found to delete, and prints a warning by
;; default.
(let ((e (make-event 'c4 'q)))
  (delete-clefs e))

=> NIL
WARNING: event::delete-clefs: no clef to delete:
[...]

;; Setting the optional WARN argument to T suppresses the warning when no clefs
;; are found.
(let ((e (make-event 'c4 'q)))
  (delete-clefs e nil))
```

```
=> NIL

;; Also returns NIL when successful
(let ((e (make-event 'c4 'q)))
  (add-clef e 'treble)
  (delete-clefs e))

=> NIL

;; Create an event, add a clef, print the MARKS-BEFORE slot, delete the event,
;; print MARKS-BEFORE again to make sure it's gone
(let ((e (make-event 'c4 'q)))
  (add-clef e 'treble)
  (print (marks-before e))
  (delete-clefs e)
  (print (marks-before e)))

=>
((CLEF TREBLE))
NIL
```

SYNOPSIS:

```
(defmethod delete-clefs ((e event) &optional (warn t) ignore1 ignore2)
```

28.5.107   event/delete-written

[event] [Methods]

DESCRIPTION:

Delete the contents of the WRITTEN-PITCH-OR-CHORD slot of a given event
object and reset to NIL.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Create an event object, print the contents of the written-pitch-or-chord ;
```

```
;; slot to see it's set to NIL, set-written to -2, print the contents of the ;
;; corresponding slot to see the data of the newly created pitch object, ;
;; delete-written, print the contents of the written-pitch-or-chord slot to see ;
;; it's empty.                             ;
(let ((e (make-event 'c4 'q)))
  (print (written-pitch-or-chord e))
  (set-written e -2)
  (print (data (written-pitch-or-chord e)))
  (delete-written e)
  (print (written-pitch-or-chord e)))

=>
NIL
BF3
NIL
```

SYNOPSIS:

```
(defmethod delete-written ((e event))
```

28.5.108   event/end-arrow

[event] [Methods]

DESCRIPTION:

 Adds an end-arrow mark to the given event object.

 NB: This method works for LilyPond only. When used with CMN, output will
     still be generated, but no mark will be added. The method prints a
     corresponding warning when applied.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;; Returns T
(let ((e (make-event 'c4 'q)))
  (end-arrow e))
```

```
=> T
WARNING:
rhythm::validate-mark: no CMN mark for END-ARROW (but adding anyway).

;; Create an event object, add end-arrow, and print the MARKS and MARKS-SLOTS
;; to see the result
(let ((e (make-event 'c4 'q)))
  (end-arrow e)
  (print (marks-before e))
  (print (marks e)))

=>
NIL
(END-ARROW)
```

SYNOPSIS:

```
(defmethod end-arrow ((e event))
```

28.5.109   event/end-trill

[event] [Methods]

DATE:

 24 Sep 2011

DESCRIPTION:

 Adds an 'end-trill-a mark to the MARKS slot of the given event object.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 T

EXAMPLE:

```
;; The end-trill method returns T
(let ((e (make-event 'c4 'q)))
```

```
  (end-trill e))

=> T

;; Add an 'end-trill-a and check the MARKS slot to see that it's there
(let ((e (make-event 'c4 'q)))
  (end-trill e)
  (marks e))

=> (END-TRILL-A)
```

SYNOPSIS:

```
(defmethod end-trill ((e event))
```

28.5.110   event/enharmonic

[event] [Methods]

DESCRIPTION:

Change the pitch of the pitch object within the given event object to its
enharmonic equivalent.

In its default form, this method only applies to note names that already
contain an indication for an accidental (such as DF4 or BS3), while
"white-key" note names (such as B3 or C4) will not produce an enharmonic
equivalent. In order to change white-key pitches to their enharmonic
equivalents, set the :force-naturals argument to T.

NB: Doesn't work on chords.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :written. T or NIL to indicate whether the test is to handle the written
   or sounding pitch in the event. T = written. Default = NIL.
 - :force-naturals. T or NIL to indicate whether to force "natural" note
   names that contain no F or S in their name to convert to their enharmonic
   equivalent (ie, B3 = CF4). NB double-flats/sharps are not implemented so
   this will only work on F/E B/C.

RETURN VALUE:

 An event object.

EXAMPLE:

```
;; The method alone returns an event object
(let ((e (make-event 'cs4 'q)))
  (enharmonic e))

=>
EVENT: start-time: NIL, end-time: NIL,
[...]

;; Create an event, change it's note to the enharmonic equivalent, and print
;; it.
(let ((e (make-event 'cs4 'q)))
  (enharmonic e)
  (data (pitch-or-chord e)))

=> DF4

;; Without the :force-naturals keyword, no "white-key" note names convert to
;; enharmonic equivalents
(let ((e (make-event 'b3 'q)))
  (enharmonic e)
  (data (pitch-or-chord e)))

=> B3

;; Set the :force-naturals keyword argument to T to enable switching white-key
;; note-names to enharmonic equivalents
(let ((e (make-event 'b3 'q)))
  (enharmonic e :force-naturals t)
  (data (pitch-or-chord e)))

=> CF4
```

SYNOPSIS:

```
(defmethod enharmonic ((e event) &key written force-naturals
                       ;; 1-based
                       chord-note-ref)
```

28.5.111   event/event-distance

[event] [Methods]

DESCRIPTION:

Get the distance (interval) in semitones between the pitch level of one
event object and a second. Negative numbers indicate direction interval
directionality. An optional argument allows distances to be always printed
as absolute values (positive).

Event-distance can also be determined between chords, in which case the
distance is measured between the highest pitch of one event object and the
lowest of the other.

ARGUMENTS:

 - A first event object.
 - A second event object.

OPTIONAL ARGUMENTS:

 - T or NIL for whether the value should be returned as an absolute
   value (i.e., always positive). Default = NIL.

RETURN VALUE:

 A number.

EXAMPLE:

```
;; The semitone distance between two single pitches in ascending direction ;
                   (let ((e1 (make-event 'c4 'q))
(e2 (make-event 'e4 'q)))
(event-distance e1 e2))

                => 4.0

;; The semitone distance between two single pitches in descending direction ;
                   (let ((e1 (make-event 'c4 'q))
(e2 (make-event 'e4 'q)))
(event-distance e2 e1))

                => -4.0
```

```
;; Set the optional argument to T to get the absolute distance (positive ;
;; number)                                          ;
                      (let ((e1 (make-event 'c4 'q))
(e2 (make-event 'e4 'q)))
(event-distance e2 e1 t))

                          => 4.0

;; The semitone distance between two chords in ascending direction ;
                      (let ((e1 (make-event '(c4 e4 g4) 'q))
(e2 (make-event '(d4 f4 a4) 'q)))
(event-distance e1 e2))

                          => 9.0
```

SYNOPSIS:

```
(defmethod event-distance ((e1 event) (e2 event) &optional absolute)
```

28.5.112   event/event-p

[event] [Functions]

DESCRIPTION:

 Test to confirm that a given object is an event object.

ARGUMENTS:

 - An object.

RETURN VALUE:

 T if the tested object is indeed an event object, otherwise NIL.

EXAMPLE:

```
;; Create an event and then test whether it is an event object
(let ((e (make-event 'c4 'q)))
  (event-p e))
=> T

;; Create a non-event object and test whether it is an event object
(let ((e (make-rhythm 4)))
  (event-p e))
```

```
=> NIL

;; The make-rest function also creates an event
(let ((e (make-rest 4)))
  (event-p e))
=> T

;; The make-punctuation-events, make-events and make-events2 functions create
;; lists of events, not events themselves.
(let ((e (make-events '((g4 q) e s))))
  (event-p e))
=> NIL
```

SYNOPSIS:

```
(defun event-p (thing)
```

28.5.113   event/flat-p

[event] [Methods]

DESCRIPTION:

 Determine whether the pitch of a given event object has a flat.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the test is to handle the written or
   sounding pitch in the event. T = written. Default = NIL.

RETURN VALUE:

 Returns T if the note tested has a flat, otherwise NIL (ie, is natural or
 has a sharp).

EXAMPLE:

```
;; Returns T when the note is flat
(let ((e (make-event 'df4 'q)))
  (flat-p e))
```

```
=> T

;; Returns NIL when the note is not flat (ie, is sharp or natural)
(let ((e (make-event 'c4 'q)))
  (flat-p e))

=> NIL

(let ((e (make-event 'cs4 'q)))
  (flat-p e))

=> NIL
```

SYNOPSIS:

```
(defmethod flat-p ((e event) &optional written)
```

28.5.114   event/flat-to-sharp

[event] [Methods]

DATE:

 Mon 10 Feb 2020 18:41:40 GMT - London

DESCRIPTION:

 Change the flat (or flats) in an event to a sharp or (sharps).

ARGUMENTS:

 An event object

OPTIONAL ARGUMENTS:

```
 - clone: clone the evnet. DEFAULT = nil
 - written: apply to written or sounding pitch? DEFAULT = nil (sounding)
```

RETURN VALUE:

 An event object

EXAMPLE:

```
(let ((e (make-event 'bf4 'q)))
      (print-simple (flat-to-sharp e)))

=> AS4 Q,
NIL

(let ((e (make-event '(as4 bf4) 'q)))
      (print-simple (flat-to-sharp e)))

=> (AS4 AS4) Q,
NIL

;;; NB This method is non-destructive
(let ((e (make-event 'bf4 'q)))
      (print-simple (flat-to-sharp e))
      (print-simple e))

=> AS4 Q,
BF4 Q,
NIL
```

SYNOPSIS:

```
(defmethod flat-to-sharp ((e event) &optional clone written)
```

28.5.115   event/force-artificial-harmonic

[event] [Methods]

DESCRIPTION:

Change the pitch-or-chord content of a given event object such that the
existing pitch will be notated as an artificial harmonic.

The method creates pitch data for an artificial harmonic that will result
in the specified pitch, rather than adding an artificial harmonic to the
specified pitch. Thus, the method changes the existing pitch content by
transposing the specified pitch down two octaves and adding a new pitch one
perfect fourth above it (changing the given pitch object to a chord
object). It then also adds the mark 'flag-head to the MARKS slot of the
upper pitch for printing layout so that the upper pitch is printed as a
diamond.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - an instrument object

RETURN VALUE:

 The chord object which creates the harmonic.

EXAMPLE:

```
;; The method returns NIL.
(let ((e (make-event 'c7 'q)))
  (force-artificial-harmonic e))
=> NIL

;; Create an event object, apply force-artificial-harmonic, then get the new
;; pitch material
(let ((e (make-event 'c7 'q)))
  (force-artificial-harmonic e)
  (loop for p in (data (pitch-or-chord e)) collect (data p)))
=> (C5 F5)

;; Create an event object, apply force-artificial-harmonic, then get the marks
;; attached to each note in the object to see the 'flag-head
(let ((e (make-event 'c7 'q)))
  (force-artificial-harmonic e)
  (loop for p in (data (pitch-or-chord e)) collect (marks p)))
=> (NIL (FLAG-HEAD))
```

SYNOPSIS:

```
(defmethod force-artificial-harmonic ((e event)
                                      &optional instrument (warn t)
                                        (naturals t) ignore)
```

28.5.116   event/force-rest

[event] [Methods]

DESCRIPTION:

 Changes a given event object to a rest by setting both the PITCH-OR-CHORD
 and WRITTEN-PITCH-OR-CHORD slots to NIL and the IS-REST slot to T.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 - An event object

EXAMPLE:

```
;; The method returns an event object.  ;
                    (let ((e (make-event 'c4 'q)))
(force-rest e))

                    =>
                    EVENT: start-time: NIL, end-time: NIL,
                    [...]

;; Create an event object, apply force-rest, then print the corresponding slots ;
;; to see the effectiveness of the method ;
                    (let ((e (make-event 'c4 'q)))
(force-rest e)
(print (pitch-or-chord e))
(print (written-pitch-or-chord e))
(print (is-rest e)))

                    =>
                    NIL
                    NIL
                    T
```

SYNOPSIS:

```
(defmethod force-rest :after ((e event))
```

28.5.117   event/get-amplitude

[event] [Methods]

DESCRIPTION:

 Return the amplitude attached to a given event object.

 An optional argument allows the amplitude to be converted to and returned
 as a MIDI value.

ARGUMENTS:

- An event object.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the amplitude value is to be returned as a
  standard digital amplitude (a number between 0.0 and 1.0) or as a
  standard MIDI velocity value (a whole number between 0 and 127). T = MIDI
  value. Default = NIL.

RETURN VALUE:

 If the optional argument is set to NIL, returns a real number.

 If the optional argument is set to T, returns a whole number (and a
 remainder).

EXAMPLE:

```
;; Get the amplitude as a decimal value. (Each new event object has a default
;; amplitude of 0.7).
(let ((e (make-event 'c4 'q)))
  (get-amplitude e))

=> 0.7

;; Get the amplitude as a rounded MIDI value.
(let ((e (make-event 'c4 'q)))
  (get-amplitude e t))

=> 89, -0.10000000000000853
```

SYNOPSIS:

```
(defmethod get-amplitude ((e event) &optional (midi nil))
```

28.5.118   event/get-clef

[event] [Methods]

DESCRIPTION:

 Return the symbol associated with the key CLEF in the MARKS-BEFORE slot of
 the given event object.

ARGUMENTS:

- An event object.

OPTIONAL ARGUMENTS:

 - (Internal "ignore" arguments only; not needed by the user).

RETURN VALUE:

 Returns the given clef name as a symbol if successful.

 Returns NIL if there is no clef name found in the MARKS-BEFORE slot of the
 given event object.

EXAMPLE:

```
;; Returns NIL when no clef is found
(let ((e (make-event 'c4 'q)))
  (get-clef e))

=> NIL

;; Returns the clef name as symbol when successful.
(let ((e (make-event 'c4 'q)))
  (add-clef e 'treble)
  (get-clef e))

=> TREBLE
```

SYNOPSIS:

```
(defmethod get-clef ((e event) &optional ignore1 ignore2 ignore3)
```

28.5.119   event/get-degree

[event] [Methods]

DESCRIPTION:

 Get the degree of the event, summing or averaging for chords.

ARGUMENTS:

 - an event object

OPTIONAL ARGUMENTS:

keyword arguments:
- :written. T or NIL to indicate whether to use the written (in the case of
  transposing instruments) or sounding pitches. T = written. Default = NIL.
- :sum. T or NIL to indicate whether to return the sum of the degrees
  instead of a list (see below). T = degrees. Default = NIL.
- :average: sim. to sum but T or NIL to indicate whether to return the
   average instead of sum.

RETURN VALUE:

 By default this returns a list (even if it's a single pitch), unless :sum T
 whereupon it will return a single value: the sum of the degrees if a chord,
 otherwise just the degree. :average T will return the average of the sum.
 A rest would return '(0) or 0.

EXAMPLE:

```
;;; NB This uses the quarter-tone scale so degrees are double what they would ;
;;;    be in the chromatic-scale.        ;
(let ((event (make-event '(cs4 d4) 'e))
      (rest (make-rest 'e)))
  (print (get-degree event))
  (print (get-degree rest))
  (print (get-degree event :average t))
  (get-degree event :sum t))
(122 124)
(0)
123.0
246
```

SYNOPSIS:

```
(defmethod get-degree ((e event) &key written sum average)
```

28.5.120   event/get-dynamic

[event] [Methods]

DESCRIPTION:

 Gets the dynamic marking attached to a given event object.

 NB: This method is similar to the event::get-dynamics method, but assumes
 that there is only one dynamic and returns that dynamic as a single symbol
 rather than a list. If the user suspects that multiple dynamics may have

somehow have been added to the MARKS slot of the event class, use
get-dynamics to obtain a list of all dynamics in that slot; otherwise, this
is the method that should be generally used.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 The symbol representing the dynamic if there is one attached to that event,
 otherwise NIL.

EXAMPLE:

```
;; The method returns just the dynamic marking from the MARKS list, as a symbol
(let ((e (make-event 'c4 'q)))
  (add-mark-once e 'ppp)
  (add-mark-once e 'pizz)
  (get-dynamic e))

=> PPP

;; The method returns NIL if there is no dynamic in the MARKS list
(let ((e (make-event 'c4 'q)))
  (add-mark-once e 'pizz)
  (get-dynamic e))

=> NIL
```

SYNOPSIS:

```
(defmethod get-dynamic ((e event))
```

28.5.121   event/get-dynamics

[event] [Methods]

DESCRIPTION:

 Get the list of dynamic marks from a given event object, assuming there are
 multiple dynamics present. If other non-dynamic events are also contained
 in the MARKS slot of the rhythm object within the given event object, these
 are disregarded and only the dynamic marks are returned.

    NB: This method is similar to the event::get-dynamic, but is intended for
        use should multiple dynamics have somehow become attached to the same
        event. The method event::get-dynamic is the method that should
        generally be used.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 A list containing the dynamics stored in the MARKS slot of the rhythm
 object within the given event object. NIL is returned if no dynamic marks
 are attached to the given event object.

EXAMPLE:

```
;; Create an event object and get the dynamics attached to that object. These
;; are NIL by default (unless otherwise specified).
(let ((e (make-event 'c4 'q)))
  (get-dynamics e))

=> NIL

;; Create an event object, add one dynamic and one non-dynamic mark, print all
;; marks, then retrieve only the dynamics.
(let ((e (make-event 'c4 'q)))
  (add-mark-once e 'ppp)
  (add-mark-once e 'pizz)
  (print (marks e))
  (get-dynamics e))

=>
(PIZZ PPP)
(PPP)

;; Should multiple dynamics have become attached to the same event object,
;; get-dynamics will return all dynamics present in the MARKS slot
(let ((e (make-event 'c4 'q)))
  (add-mark-once e 'pizz)
  (add-mark-once e 'ppp)
  (push 'fff (marks e))
  (print (marks e))
  (get-dynamics e))

=> (FFF PPP)
```

SYNOPSIS:

```
(defmethod get-dynamics ((e event) &optional hairpins)
```

28.5.122   event/get-frequency

[event] [Methods]

DATE:

 May 26th 2016, Edinburgh

DESCRIPTION:

 Return the frequency of the event's pitch, or a list of frequencies if it's
 a chord.

ARGUMENTS:

 - the event object

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :written. T or NIL to indicate whether to use the written (in the case of
  transposing instruments) or sounding pitches. T = written. Default = NIL.
- :force-list. T or NIL to force a single pitch's frequency into a
  list. Default = NIL.
- :average. T or NIL to make chords average their frequency.
```

RETURN VALUE:

 A single frequency (float) or list of frequencies in the case of a chord.

SYNOPSIS:

```
(defmethod get-frequency ((e event) &key written force-list average)
```

28.5.123   event/get-midi-channel

[event] [Methods]

DESCRIPTION:

Retrieve the value set for the midi-channel slot of the pitch object within
a given event object. Beware that if the pitch-or-chord slots is a chord, we
will merely return the channel of the first pitch in the chord.

ARGUMENTS:

 - An event object.

RETURN VALUE:

 An integer representing the given midi-channel value or NIL if there isn't
 one.

EXAMPLE:

```
;; The default midi-channel value for a newly created event-object is NIL
;;; unless otherwise specified.
(let ((e (make-event 'c4 'q)))
  (get-midi-channel e))

=> NIL

;; Create an event object, set its MIDI-channel and retrieve it
(let ((e (make-event 'c4 'q)))
  (set-midi-channel e 11 12)
  (get-midi-channel e))

=> 11
```

SYNOPSIS:

```
(defmethod get-midi-channel ((e event))
```

28.5.124   event/get-pitch-symbol

[event] [Methods]

DESCRIPTION:

 Retrieve the pitch symbol (CM/CMN note-name notation) of a given event
 object. Returns a single symbol if the given event object consists of a
 single pitch; otherwise, returns a list of pitch symbols if the given event
 object consists of a chord.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the test is to handle the event's
   written or sounding pitch. T = written. Default = T.

RETURN VALUE:

 A symbol, if the event object consists of only a single pitch, otherwise a
 list of pitch symbols if the event object consists of a chord.

EXAMPLE:

```
;; Get the pitch symbol of an event object with a single pitch
(let ((e (make-event 'c4 'q)))
  (get-pitch-symbol e))

=> C4

;; Getting the pitch symbol of an event object that consists of a chord returns
;; a list of pitch symbols
(let ((e (make-event '(c4 e4 g4) 'q)))
  (get-pitch-symbol e))

=> (C4 E4 G4)
```

SYNOPSIS:

```
(defmethod get-pitch-symbol ((e event) &optional (written t))
```

28.5.125   event/has-hairpin

[event] [Methods]

DESCRIPTION:

 Return T or NIL depending on whether the event has a beginning or ending
 hairpin (cresc. or dim. line).

ARGUMENTS:

 - an event object

RETURN VALUE:

 The hairpin mark the event has, as a list, or NIL if it has none.

SYNOPSIS:

(defmethod has-hairpin ((e event)))

28.5.126   event/has-mark-before

[event] [Methods]

DESCRIPTION:

 Determine whether a specifed event object has a specified mark in its
 MARKS-BEFORE slot.

ARGUMENTS:

 - An event object.
 - A mark.

RETURN VALUE:

 Returns the specified mark if the mark exists in the MARKS-BEFORE slot,
 otherwise returns NIL

EXAMPLE:

```
;;; Returns the specified mark if that mark is present
(let ((e (make-event 'c4 4)))
  (add-mark-before e 'ppp)
  (has-mark-before e 'ppp))

=> (PPP)

;;; Returns NIL if the specified mark is not present
(let ((e (make-event 'c4 4)))
  (add-mark-before e 'ppp)
  (has-mark-before e 'fff))

=> NIL
```

SYNOPSIS:

(defmethod has-mark-before ((e event) mark &optional (test #'equal)))

28.5.127   event/highest

[event] [Methods]

DESCRIPTION:

Get the highest pitch (of a chord) in a given event object. If the given
event object contains a single pitch only, that pitch is returned.

ARGUMENTS:

- An event object.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
;; Returns a pitch object
(let ((e (make-event 'c4 'q)))
  (highest e))

=>
PITCH: frequency: 261.6255569458008, midi-note: 60, midi-channel: NIL
pitch-bend: 0.0
degree: 120, data-consistent: T, white-note: C4
nearest-chromatic: C4
src: 1.0, src-ref-pitch: C4, score-note: C4
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: NIL, natural: T,
octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
show-accidental: T, white-degree: 28,
accidental: N,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4

;; Returns the highest note of a chord object within an event object ;
                    (let ((e (make-event '(d4 fs4 a4) 'q)))
(data (highest e)))

=> A4
```

SYNOPSIS:

```
(defmethod highest ((e event))
```

28.5.128   event/inc-duration

[event] [Methods]

DESCRIPTION:

Increase the duration of a given event object by a specified time in
seconds. This will also result in new values for the end-time,
duration-in-tempo, compound-duration, and compound-duration-in-tempo slots.

NB: Changing this value directly could result in incorrect timing info in a
    bar. Also, the *-in-duration slots are changed by the same amount as the
    other slots, i.e. tempo does not play a role here

ARGUMENTS:

- An event object.
- A value that is the increment in seconds by which the duration is to be
  extended.

RETURN VALUE:

The new duration in seconds.

EXAMPLE:

```
;;; Create a slippery-chicken object, assign a variable to one of the event
;;; objects it contains, print the corresponding duration slots; apply
;;; inc-duration and print the corresponding duration slots again to see the
;;; change.
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vc (cello :midi-channel 1))))
         :set-palette '((1 ((gs3 as3 b3))))
         :set-map '((1 (1)))
         :rthm-seq-palette '((1 ((((2 4) q (e) s (32) 32))
                                 :pitch-seq-palette ((1 2 3)))))
         :rthm-seq-map '((1 ((vc (1)))))))
       (e (get-event mini 1 3 'vc)))
  (print (end-time e))
```

```
  (print (duration-in-tempo e))
  (print (compound-duration-in-tempo e))
  (inc-duration e 7.0)
  (print (end-time e))
  (print (duration-in-tempo e))
  (print (compound-duration-in-tempo e)))

=>
1.75
0.25
0.25
8.75
7.25
7.25
```

SYNOPSIS:

```
(defmethod inc-duration ((e event) inc)
```

28.5.129   event/is-chord

[event] [Methods]

DESCRIPTION:

 Test to determine whether a given event object consists of a chord (as
 opposed to a single pitch or a rest).

ARGUMENTS:

 - An event object.

RETURN VALUE:

 - If the given event object is a chord, the method returns a number that is
   the number of notes in the chord.
 - Returns NIL if the given event object is not a chord.

EXAMPLE:

```
;; Returns NIL if not a chord            ;
(let ((e (make-event 'c4 'q)))
  (is-chord e))

=> NIL
```

```
;; If a chord, returns the number of notes in the chord ;
(let ((e (make-event '(c4 e4 g4) 'q)))
  (is-chord e))

=> 3

;; A rest is not a chord                  ;
(let ((e (make-rest 'q)))
  (is-chord e))

=> NIL
```

SYNOPSIS:

```
(defmethod is-chord ((e event))
```

28.5.130   event/is-dynamic

[event] [Functions]

DESCRIPTION:

Determine whether a specified symbol belongs to the list of predefined
dynamic marks.

ARGUMENTS:

 - A symbol.

RETURN VALUE:

NIL if the specified mark is not found on the predefined list of possible
dynamic marks, otherwise the tail of the list of possible dynamics starting
with the given dynamic.

EXAMPLE:

```
(is-dynamic 'pizz)
=> NIL

(is-dynamic 'f)
=> (F FF FFF FFFF)
```

SYNOPSIS:

```
(defun is-dynamic (mark)
```

28.5.131   event/is-single-pitch

[event] [Methods]

DESCRIPTION:

Test to see if an event object consists of a single pitch (as opposed to a chord or a rest).

ARGUMENTS:

- An event object.

RETURN VALUE:

Returns T if the given event object consists of a single pitch, otherwise returns NIL.

EXAMPLE:

```
;; Returns T if the event object consists of a single pitch ;
(let ((e (make-event 'c4 'q)))
  (is-single-pitch e))

=> T

;; Returns NIL if the event object is a chord ;
(let ((e (make-event '(c4 e4 g4) 'q)))
  (is-single-pitch e))

=> NIL

;; Also returns NIL if the event object is a rest ;
(let ((e (make-rest 'q)))
  (is-single-pitch e))

=> NIL
```

SYNOPSIS:

```
(defmethod is-single-pitch ((e event))
```

28.5.132   event/lowest

[event] [Methods]

DESCRIPTION:

Get the lowest pitch (of a chord) in a given event object. If the given
event object contains a single pitch only, that pitch is returned.

ARGUMENTS:

- An event object.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
;; Returns a pitch object                 ;
(let ((e (make-event 'c4 'q)))
  (lowest e))

=>
PITCH: frequency: 261.6255569458008, midi-note: 60, midi-channel: NIL
pitch-bend: 0.0
degree: 120, data-consistent: T, white-note: C4
nearest-chromatic: C4
src: 1.0, src-ref-pitch: C4, score-note: C4
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: NIL, natural: T,
octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
show-accidental: T, white-degree: 28,
accidental: N,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4

;; Returns the lowest note of a chord object within an event object ;
(let ((e (make-event '(d4 fs4 a4) 'q)))
  (data (lowest e)))

=> D4
```

SYNOPSIS:

```
(defmethod lowest ((e event))
```

28.5.133   event/make-event

[event] [Functions]

DESCRIPTION:

 Create an event object for holding rhythm, pitch, and timing data.

ARGUMENTS:

- A pitch or chord. This can be one of those objects (will be added to the
  pitch-or-chord slot without cloning), or a pitch symbol or list of pitch
  symbols (for a chord).
- The event's rhythm (e.g. 'e). If this is a number, its interpretation is
  dependent on the value of duration (see below). NB if this is a rhythm
  object, it will be cloned.

OPTIONAL ARGUMENTS:

 keyword arguments:
- :start-time. The start time of the event in seconds. Default = NIL.
- :is-rest. Set to T or NIL to indicate whether or not the given event is a
  rest. Default = NIL. NB: The make-rest method is better suited to making
  rests; however, if using make-event to do so, the pitch-or-chord slot
  must be set to NIL.
- :is-tied-to. This argument is for score output and playing purposes. Set
  to T or NIL to indicate whether this event is tied to the previous event
  (i.e. it won't sound independently). Default = NIL.
- :duration. T or NIL to indicate whether the specified duration of the
  event has been stated in absolute seconds, not a known rhythm like
  'e. Thus (make-event 'c4 4 :duration nil) indicates a quarter note with
  duration 1, but (make-event '(c4 d4) 4 :duration t) indicates a whole
  note with an absolute duration of 4 seconds (both assuming a tempo of
  60). Default = NIL.
- :amplitude sets the amplitude of the event. Possible values span from 0.0
  (silent) to maximum of 1.0. Default = (get-sc-config 'default-amplitude).
- :tempo. A number to indicate the tempo of the event as a normal bpm
  value. Default = 60. This argument is only used when creating the rhythm
  from a given duration rather than from a letter value.
- :tempo-change. A number or tempo-object to attach a tempo change to the
  event. This is the argument you need to set in order to change playback
  speed.
- :midi-channel. A number from 0 to 127 indicating the MIDI channel on
  which the event should be played back. Default = NIL.
- :microtones-midi-channel. If the event is microtonal, this argument
  indicates the MIDI-channel to be used for the playback of the microtonal

   notes. Default = NIL. NB: See player.lsp/make-player for details on
   microtones in MIDI output.
- :transposition. A number in semitones that indicates the transposition of
   the instrument that this event is being created for. E.g. -2 would be
   for a B-flat clarinet.
- :written. The given pitch or chord is the written value. In this case
   the sounding value will be set according to the (required) transposition
   argument. Default = NIL.
- :bar-num. Set the bar-num slot of the event. Usually maintained by various
   classes that contain events. Default = -1.

RETURN VALUE:

- An event object.

EXAMPLE:

```
;; A quarter-note (crotchet) C
(make-event 'c4 4)
=>
EVENT: start-time: NIL, end-time: NIL,
duration-in-tempo: 0.0,
compound-duration-in-tempo: 0.0,
amplitude: 0.7,
bar-num: -1, marks-before: NIL,
tempo-change: NIL
instrument-change: NIL
display-tempo: NIL, start-time-qtrs: -1,
midi-time-sig: NIL, midi-program-changes: NIL,
8va: 0
pitch-or-chord:
PITCH: frequency: 261.6255569458008, midi-note: 60, midi-channel: NIL
pitch-bend: 0.0
degree: 120, data-consistent: T, white-note: C4
nearest-chromatic: C4
src: 1.0, src-ref-pitch: C4, score-note: C4
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: NIL, natural: T,
octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
show-accidental: T, white-degree: 28,
accidental: N,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4
```

```
written-pitch-or-chord: NIL
RHYTHM: value: 4.0, duration: 1.0, rq: 1, is-rest: NIL, score-rthm: 4.0f0,
undotted-value: 4, num-flags: 0, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 1.0, is-grace-note: NIL,
needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 4,
tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 4, tag: NIL,
data: 4

;; Create a whole-note (semi-breve) chord, then print its data, value, duration
;; and pitch content
(let ((e (make-event '(c4 e4 g4) 4 :duration t)))
  (print (data e))
  (print (value e))
  (print (duration e))
  (print (loop for p in (data (pitch-or-chord e)) collect (data p))))

=>
W
1.0f0
4.0
(C4 E4 G4)

;; Create a single-pitch quarter-note event which is tied to, plays back on
;; MIDI channel 1 and has an amplitude of 0.5, then print these values by
;; accessing the corresponding slots.
(let ((e (make-event 'c4 4
                     :is-tied-to t
                     :midi-channel 1
                     :amplitude 0.5)))
  (print (is-tied-to e))
  (print (midi-channel (pitch-or-chord e)))
  (print (amplitude e)))

=>
T
1
0.5

;; Create an event object that consists of a quarter-note rest and print the
;; contents of the corresponding slots
(let ((e (make-event nil 'q :is-rest t)))
  (print (pitch-or-chord e))
  (print (data e))
```

```
  (print (is-rest e)))

=>
NIL
Q
T
```

SYNOPSIS:

```
(defun make-event (pitch-or-chord rthm
                   &key
                     start-time
                     is-rest
                     is-tied-to
                     ;; MDE Fri Jan  5 10:38:22 2018 -- added
                     is-tied-from
                     duration
                     ;; MDE Wed Jun 15 20:39:05 2016 -- defaults to 1 and 2
                     (midi-channel 1)
                     (microtones-midi-channel 2)
                     ;; MDE Thu May 31 19:03:59 2012 -- allow us to auto-set the
                     ;; written-pitch-or-chord slot
                     transposition
                     ;; MDE Sat Apr 20 15:13:41 2013 -- allow us to create
                     ;; written pitch events and auto-set sounding
                     written
                     (amplitude (get-sc-config 'default-amplitude))
                     ;; MDE Thu Jun 16 13:05:21 2016
                     tempo-change
                     ;; MDE Fri Nov  2 16:37:05 2018
                     (bar-num -1)
                     (tempo 60))
```

28.5.134   event/make-events

[event] [Functions]

DESCRIPTION:

 Make a list of events using the specified data, whereby a list indicates a
 note (or chord) and its rhythm and a single datum is the rhythm of a rest.

ARGUMENTS:

 - A list of symbols and/or sublists; see below for examples.

OPTIONAL ARGUMENTS:

- A whole number indicating the MIDI channel on which the event is to be
  played.
- A whole number indicating the MIDI channel on which microtonal pitches of
  the event are to be played. NB: See player.lsp/make-player for details on
  microtones in MIDI output.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; Create a list of events including a quarter note, two rests, and a chord,
;; then print-simple its contents
(let ((e (make-events '((g4 q) e s ((d4 fs4 a4) s)))))
  (loop for i in e do (print-simple i)))
=>
G4 Q, rest E, rest S, (D4 FS4 A4) S,

;; Create a list of events to be played on MIDI-channel 3, then check the MIDI
;; channels of each sounding note
(let ((e (make-events '((g4 q) e s (a4 s) q e (b4 s)) 3)))
  (loop for i in e
     when (not (is-rest i))
     collect (midi-channel (pitch-or-chord i))))
=> (3 3 3)
```

SYNOPSIS:

```
(defun make-events (data-list &optional (midi-channel 1)
                                        (microtones-midi-channel 2))
```

28.5.135   event/make-events2

[event] [Functions]

DESCRIPTION:

 Like make-events, but rhythms and pitches are given in separate lists to
 allow for rhythms with ties using "+" etc. "Nil" or "r" given in the pitch
 list indicates a rest; otherwise, a single note name will set a single
 pitch while a list of note names will set a chord. Pitches for tied notes
 only have to be given once.

ARGUMENTS:

- A list of rhythms.
- A list of note names (including NIL or R for rests).

OPTIONAL ARGUMENTS:

- A whole number value to indicate the MIDI channel on which to play back
  the event.
- A whole number value to indicate the MIDI channel on which to play back
  microtonal pitch material for the event. NB: See player.lsp/make-player
  for details on microtones in MIDI output.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; Create a make-events2 list and use the print-simple function to retrieve its
;; contents.
(let ((e (make-events2 '(q e e. h+s 32 q+te) '(cs4 d4 (e4 g4 b5) nil a3 r))))
  (loop for i in e do (print-simple i)))
=>
CS4 Q, D4 E, (E4 G4 B5) E., rest H, rest S, A3 32, rest Q, rest TE,

;; Create a list of events using make-events2, indicating they be played back
;; on MIDI-channel 3, then print the corresponding slots to check it
(let ((e (make-events2 '(q e. h+s 32 q+te) '(cs4 b5 nil a3 r) 3)))
  (loop for i in e
     when (not (is-rest i))
     collect (midi-channel (pitch-or-chord i))))
=>
(3 3 3)
```

SYNOPSIS:

```
(defun make-events2 (rhythms pitches
                     &optional (midi-channel 1) (microtones-midi-channel 2))
```

28.5.136   event/make-events3

[event] [Functions]

DATE:

September 12th 2020, Heidhausen

DESCRIPTION:

Make a list of events using various approaches. Either with separate lists
of pitches and rhythms make-events and make-events2 will be called according
to the type of arguments given. Note that when specifying pitches, octaves
don't have to be retyped if they don't change.

ARGUMENTS:

Two required arguments but their type and function vary according to
requirements. See examples below.

OPTIONAL ARGUMENTS:

- A whole number value to indicate the MIDI channel on which to play back
  the event.
- A whole number value to indicate the MIDI channel on which to play back
  microtonal pitch material for the event. NB: See player.lsp/make-player
  for details on microtones in MIDI output.

RETURN VALUE:

a list of events

EXAMPLE:

```
;; simple list of pitch/rhythm pairs but no chords or rests (so no need for list
;; pairs)
(make-events3 '(g4 s a s b 32 c 32) nil)
;; pitch and rhythms in pairs in a single list, including chords and rests
(make-events3 '((g4 q) e s ((d4 fs4 a4) s)) nil)
;; using separate lists for rhythms and pitches, tied rhythms, single pitches, a
;; chord, and both nil and r to indicate rests
(make-events3 '(q e e. h+s 32 q+te) '(cs4 d4 (e4 g4 b5) nil a3 r))
;; just one pitch but various rhythms (but here we can't use tied rhythms)
(make-events3 '(q e e. s e s s s s) 'cs6)
;; just one rhythm but various pitches/chords and all in octave 4 without having
;; to retype (here we can't use tied rhythms either)
(make-events3 's '(c4 d e (f a) r bf)) ; r = rest here too
```

SYNOPSIS:

```
(defun make-events3 (data1 data2 &key (midi-channel 1)
                                      (microtones-midi-channel 2))
```

28.5.137   event/make-punctuation-events

[event] [Functions]

DESCRIPTION:

Given a list of numbers, a rhythm, and a note name or list of note names,
create a new list of single events separated by rests.

The rhythm specified serves as the basis for the new list. The numbers
specified represent groupings in the new list that are each made up of one
rhythm followed by rests. Each consecutive grouping in the new list has the
length of each consecutive number in the numbers list multiplied by the
rhythm specified.

Notes can be a single note or a list of notes. If the latter, they'll be
used one after the other, repeating the final note once reached.

ARGUMENTS:

- A list of grouping lengths.
- A rhythm.
- A note name or list of note names.

RETURN VALUE:

A list.

EXAMPLE:

```
;; Create a list of three groups that are 2, 3, and 5 16th-notes long, with the
;; first note of each grouping being a C4, then print-simple it's contents.
(let ((pe (make-punctuation-events '(2 3 5) 's 'c4)))
  (loop for e in pe do (print-simple e)))
=>
C4 S, rest S, C4 S, rest S, rest S, C4 S, rest S, rest S, rest S, rest S,

;; Create a list of "punctuated" events using a list of note names. Once the
;; final note name is reached, it is repeated for all remaining non-rest
;; rhythms.
(let ((pe (make-punctuation-events '(2 3 5 8) 'q '(c4 e4))))
  (loop for e in pe do (print-simple e)))
=>
C4 Q, rest Q, E4 Q, rest Q, rest Q, E4 Q, rest Q, rest Q, rest Q, rest Q, E4 Q,
rest Q, rest Q, rest Q, rest Q, rest Q, rest Q, rest Q,
```

SYNOPSIS:

```
(defun make-punctuation-events (distances rhythm notes)
```

28.5.138   event/make-rest

[event] [Functions]

DESCRIPTION:

 Create an event object that consists of a rest.

ARGUMENTS:

 - A rhythm (duration).

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start-time. A number that is the start-time of the event in seconds.
 - :duration. T or NIL. T indicates that the duration given is a value of
   absolute seconds rather than a known rhythm (e.g. 'e). Default = NIL.
 - :tempo. Beats per minute. Note that this slot is used for duration
   calculations. If you're making events 'by hand' and you want a new tempo to
   be written to MIDI files you'll need to set the tempo-change slot with
   either a number or tempo object. If you then want the tempo to be written
   to the score, set display-tempo to T. Default = 60.

RETURN VALUE:

 - An event object.

EXAMPLE:

```
;; Make an event object consisting of a quarter rest
(make-rest 4)
=>
EVENT: start-time: NIL, end-time: NIL,
duration-in-tempo: 0.0,
compound-duration-in-tempo: 0.0,
amplitude: 0.7,
bar-num: -1, marks-before: NIL,
tempo-change: NIL
instrument-change: NIL
display-tempo: NIL, start-time-qtrs: -1,
```

```
midi-time-sig: NIL, midi-program-changes: NIL,
8va: 0
pitch-or-chord: NIL
written-pitch-or-chord: NIL
RHYTHM: value: 4.0, duration: 1.0, rq: 1, is-rest: T, score-rthm: 4.0f0,
undotted-value: 4, num-flags: 0, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 1.0, is-grace-note: NIL,
needs-new-note: NIL, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 4,
tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 4, tag: NIL,
data: 4

;; Make an event object consisting of 4 seconds of rest (rather than a quarter
;; indicated by the :duration t) starting at time-point 13.7 seconds, then
;; print the corresponding slot values.
(let ((e (make-rest 4 :start-time 13.7 :duration t)))
  (print (is-rest e))
  (print (data e))
  (print (duration e))
  (print (value e))
  (print (start-time e)))
=>
T
W
4.0
1.0f0
13.7
```

SYNOPSIS:

```
(defun make-rest (rthm &key start-time duration (tempo 60))
```

28.5.139   event/microtonal

[event] [Methods]

DATE:

 January 30th 2017

DESCRIPTION:

 Return numerical information about the microtonality of an event,
 essentially how many microtonal pitches there are.

ARGUMENTS:

 - the event object

OPTIONAL ARGUMENTS:

 T or NIL to determine whether numbers should always be returned (i.e. zeros
 when there are no microtonal pitches or any pitches at all in the case of
 rests)

RETURN VALUE:

 NIL if the event is a rest (or there are no microtones and <force-number>
 is nil), otherwise two values: the number of microtonal pitches in the
 event (0, 1, or more if it's a chord), the number of pitches in the event
 (2+ for chords otherwise 0 for a rest and 1 for a non-chord).

EXAMPLE:

```
(microtonal (make-event 'cs4 'q))
NIL

(microtonal (make-event 'cs4 'q) t)
0
1

(microtonal (make-event 'cqs4 'q) t)
1
1

(microtonal (make-event 'cqs4 'q))
1
1

(microtonal (make-event '(cqs4 e4 g4) 'q))
1
3

(microtonal (make-event nil 'q))
NIL

(microtonal (make-event nil 'q) t)
0
1
```

SYNOPSIS:

```
(defmethod microtonal ((e event) &optional force-number)
```

28.5.140   event/midi-control-change

[event] [Methods]

DATE:

 August 21st 2019, Heidhausen

DESCRIPTION:

 Add a MIDI control change to the midi-control-changes list slot.

ARGUMENTS:

```
 - the event
 - the MIDI channel (integer)
 - the MIDI channel (integer)
 - the control change value (integer)
```

EXAMPLE:

```
;;; sustain pedal down
(midi-control-change (make-event 'c4 'q) 1 64 127)
;;; sustain pedal half down
(midi-control-change (make-event 'c4 'q) 1 64 63)
;;; sustain pedal up
(midi-control-change (make-event 'c4 'q) 1 64 0)
;;; una corda
(midi-control-change (make-event 'c4 'q) 1 67 127)
;;; tre corde
(midi-control-change (make-event 'c4 'q) 1 67 0)
;;; sost. (middle) pedal down
(midi-control-change (make-event 'c4 'q) 1 66 127)
```

RETURN VALUE:

 The list of control-changes, including the new ones

SYNOPSIS:

```
(defmethod midi-control-change ((e event) channel controller-number value)
```

28.5.141   event/natural-p

[event] [Methods]

DESCRIPTION:

Determine whether the pitch of a given event object is a natural note (no sharps or flats).

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the test is to handle the written or
   sounding pitch in the event. T = written. Default = NIL.

RETURN VALUE:

Returns T if the note tested is natural, otherwise NIL (ie, has a flat or has a sharp).

EXAMPLE:

```
;; Returns T when the note is natural
(let ((e (make-event 'c4 'q)))
  (natural-p e))

=> T

;; Returns NIL when the note is not natural (ie, is sharp or flat)
(let ((e (make-event 'cs4 'q)))
  (natural-p e))

=> NIL

(let ((e (make-event 'df4 'q)))
  (natural-p e))

=> NIL
```

SYNOPSIS:

```
(defmethod natural-p ((e event) &optional written)
```

28.5.142   event/no-accidental

[event] [Methods]

DESCRIPTION:

Sets the SHOW-ACCIDENTAL and ACCIDENTAL-IN-PARENTHESES slots of the given
event object to NIL.

ARGUMENTS:

- An event object.

RETURN VALUE:

Always returns NIL.

EXAMPLE:

```
;; The SHOW-ACCIDENTAL slot is automatically set to T on new event objects
;; that consist of a sharp or flat note.
(let ((e (make-event 'cs4 'q)))
  (show-accidental (pitch-or-chord e)))

=> T

;; The method no-accidental sets the SHOW-ACCIDENTAL slot to NIL (and the
;; ACCIDENTAL-IN-PARENTHESES if not already).
(let ((e (make-event 'cs4 'q)))
  (no-accidental e)
  (show-accidental (pitch-or-chord e)))

=> NIL
```

SYNOPSIS:

```
(defmethod no-accidental ((e event))
```

28.5.143   event/num-notes

[event] [Methods]

DATE:

November 6th 2018, Heidhausen

DESCRIPTION:

 Return the number of notes in the event object i.e. the number of notes in
 the chord, 1 if it's a single pitch or 0 if it's a rest.

ARGUMENTS:

 - the event object

RETURN VALUE:

 An integer value for the number of notes in the event.

SYNOPSIS:

(defmethod num-notes ((e event))

28.5.144   event/output-midi

[event] [Methods]

DESCRIPTION:

 Generate the data necessary for MIDI output for a given event object. Note
 that for events that contain chords, the event's amplitude slot will be used
 for all pitches unless each individual pitch object in the chord has its
 amplitude slot set to a number.

 NB: The given event object must contain data for start-time and
     midi-channel.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - A decimal number that is the number of seconds to offset the timing of
   the MIDI output.
 - A decimal number or function that is to override any other existing event
   object data for amplitude.

RETURN VALUE:

 Returns the data required for MIDI output.

EXAMPLE:

```
;; Simple use
(let ((e (make-event 'c4 'q
                     :start-time 0.0
                     :midi-channel 1)))
  (output-midi e))

=> (#i(midi time 0.0 keynum 60 duration 1.0 amplitude 0.7 channel 0))

;; Specifying time offset and forced amplitude value
(let ((e (make-event 'c4 'q
                     :start-time 0.0
                     :midi-channel 1)))
  (output-midi e 0.736 0.3))

=> (#i(midi time 0.736 keynum 60 duration 1.0 amplitude 0.3 channel 0))
```

SYNOPSIS:

```
(defmethod output-midi ((e event) &optional (time-offset 0.0) force-velocity)
```

28.5.145   event/pitch-

[event] [Methods]

DESCRIPTION:

Determine the interval in half-steps between two pitches.

NB: This is determined by subtracting the MIDI note value of one event from
the other. Negative numbers may result if the greater MIDI note value is
subtracted from the lesser.

ARGUMENTS:

- A first event object.
- A second event object.

RETURN VALUE:

A number.

EXAMPLE:

```
(let ((e1 (make-event 'c4 'q))
      (e2 (make-event 'a3 'q)))
  (pitch- e1 e2))

=> 3.0

;; Subtracting the upper from the lower note returns a negative number
(let ((e1 (make-event 'a3 'q))
      (e2 (make-event 'c4 'q)))
  (pitch- e1 e2))

=> -3.0
```

SYNOPSIS:

```
(defmethod pitch- ((e1 event) (e2 event))
```

28.5.146   event/pitch-or-chord=

[event] [Methods]

DATE:

 Tue 29 Oct 2019 08:59:59 GMT - Warwick

DESCRIPTION:

 Convenience method, test to see if the pitch-or-chord slots of two event
 objects are the same.

ARGUMENTS:

 - a pitch or chord object
 - a pitch or chord object

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not enharmonic pitches are considered
   equal. T = enharmonic pitches are considered equal. Default = NIL.
 - a number to indicate the frequency deviation allowed before returning NIL.

RETURN VALUE:

 T if the values of the two specified pitch or chord objects are equal, otherwise
 NIL.

EXAMPLE:

```
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 'C4)))
  (pitch-or-chord= p1 p2))

=> T

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2))

=> NIL

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2 t))

=> T

(let ((p1 (make-pitch 'c4))
      (c1 (make-chord '(c4 e4 g4))))
  (pitch-or-chord= p1 c1))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(c4 e4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs4 ff4 g4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs3 ff4 g4))))
  (pitch-or-chord= c1 c2 t))

=> T
```

SYNOPSIS:

```
(defmethod pitch-or-chord= ((e1 event) (e2 event)
                            &optional enharmonics-are-equal
                              (frequency-tolerance 0.01))
```

28.5.147   event/remove-dynamics

[event] [Methods]

DESCRIPTION:

Remove all dynamic symbols from the list of marks attached to a given event
object.

NB: This doesn't change the amplitude.

ARGUMENTS:

- An event object.

OPTIONAL ARGUMENTS:

- T or NIL to remove hairpin marks too (e.g. cresc-beg). Default = NIL.

RETURN VALUE:

Returns the modified list of marks (which could be empty) attached to the
given event object if the specified dynamic was initially present in that
list and successfully removed, otherwise returns NIL.

EXAMPLE:

```
;; Create an event object, add one dynamic mark and one non-dynamic mark, print
;; all marks attached to the object, and remove just the dynamics from that
;; list of all marks.
(let ((e (make-event 'c4 'q)))
  (add-mark-once e 'ppp)
  (add-mark-once e 'pizz)
  (print (marks e))
  (remove-dynamics e))

=>
(PIZZ PPP)
(PIZZ)

;; Attempting to remove dynamics when none are present returns NIL.
```

```
(let ((e (make-event 'c4 'q)))
  (remove-dynamics e))
```

```
=> NIL
```

SYNOPSIS:

```
(defmethod remove-dynamics ((e event) &optional hairpins)
```

28.5.148   event/replace-mark

[event] [Methods]

DESCRIPTION:

Replace a specified mark of a given event object with a second specified
mark. If an event object contains more than one mark, individual marks can
be changed without modifying the remaining marks.

ARGUMENTS:

```
- An event object.
- The mark to be replaced.
- The new mark.
```

OPTIONAL ARGUMENTS:

```
- T or NIL to indicate whether the mark to be replaced is in the
  MARKS-BEFORE slot. T = it is in the MARKS-BEFORE slot. Default = NIL.
```

RETURN VALUE:

Returns the new value of the MARKS/MARKS-BEFORE slot of the given object.

EXAMPLE:

```
;; Add marks to the MARKS slot and replace 'a with 'batt ;
                  (let ((e (make-event 'c4 'q)))
(loop for m in '(a s pizz)
do (add-mark e m))
(replace-mark e 'a 'batt))

                => (PIZZ S BATT)
```

```
;; Add marks to the MARKS-BEFORE slot and replace 'arco with 'at ;
                      (let ((e (make-event 'c4 'q)))
(loop for m in '(arco col-legno)
do (add-mark-before e m))
(replace-mark e 'arco 'at t))
```

```
                      => (COL-LEGNO AT)
```

```
                      |#
(defmethod replace-mark ((e event) what with &optional before)
```

28.5.149   event/round-to-nearest

[event] [Methods]

DESCRIPTION:

 Round the written and sounding pitch/chord objects to the nearest in the
 current or given scale.

ARGUMENTS:

 the event object

OPTIONAL ARGUMENTS:

 keyword argument:
 :scale. The scale to use when rounding. (Common Music tuning object or
 symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or 'quarter-tone
 only at present. Default is the current scale as set by (in-scale :...).

RETURN VALUE:

 the modified event object

SYNOPSIS:

```
(defmethod round-to-nearest ((e event) &key (scale cm::*scale*))
```

28.5.150   event/set-midi-channel

[event] [Methods]

DESCRIPTION:

Set the MIDI-channel and microtonal MIDI-channel for the pitch object
within a given event object.

ARGUMENTS:

 - An event object.
 - A whole number indicating the MIDI-channel to be used for playback of
   this event object.
 - A whole number indicating the MIDI-channel to be used for playback of the
   microtonal pitch material of this event.

RETURN VALUE:

 Returns the value of the MIDI-channel setting (a whole number) if the
 MIDI-channel slot has been set, otherwise NIL.

EXAMPLE:

```
;; Unless specified the MIDI channel of a newly created event object defaults
;;; to NIL.
(let ((e (make-event 'c4 'q)))
  (midi-channel (pitch-or-chord e)))

=> NIL

(let ((e (make-event 'c4 'q)))
  (set-midi-channel e 7 8)
  (midi-channel (pitch-or-chord e)))

=> 7
```

SYNOPSIS:

```
(defmethod set-midi-channel ((e event) midi-channel
                              &optional microtonal-midi-channel)
```

28.5.151   event/set-midi-time-sig

[event] [Methods]

DESCRIPTION:

 Sets a MIDI time signature for the given event object. This must be a
 time-sig object, not just a time signature list.

ARGUMENTS:

 - An event object.
 - A time-sig object.

RETURN VALUE:

 Returns a time-sig object.

EXAMPLE:

```
;; Creating a new event object sets the midi-time-sig slot to NIL by default
(let ((e (make-event 'c4 'q)))
  (midi-time-sig e))

=> NIL

;; The set-midi-time-sig method returns a time-sig object
(let ((e (make-event 'c4 'q)))
  (set-midi-time-sig e (make-time-sig '(3 4))))

=>
TIME-SIG: num: 3, denom: 4, duration: 3.0, compound: NIL, midi-clocks: 24,
num-beats: 3
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0304", tag: NIL,
data: (3 4)

;; Set the midi-time-sig slot and read the data of the given time-sig object
(let ((e (make-event 'c4 'q)))
  (set-midi-time-sig e (make-time-sig '(3 4)))
  (data (midi-time-sig e)))

=> (3 4)
```

SYNOPSIS:

```
(defmethod set-midi-time-sig ((e event) time-sig)
```

28.5.152   event/set-written

[event] [Methods]

DESCRIPTION:

Set the written pitch (as opposed to sounding; i.e., for transposing
instruments) of a given event object. The sounding pitch remains unchanged
as a pitch object in the PITCH-OR-CHORD slot, while the written pitch is
added as a pitch object to the WRITTEN-PITCH-OR-CHORD slot.

ARGUMENTS:

 - An event object.
 - A number indicating the difference in semitones (positive or
   negative) between the written and sounding pitches. E.g. to set the
   written-note for a B-flat Clarinet, this would be 2, for an E-flat
   Clarinet, it would be -3.

RETURN VALUE:

 The 'written' pitch object.

EXAMPLE:

```
;; Returns a pitch object (here for example for a D Trumpet or Clarinet) ;
(let ((e (make-event 'c4 'q)))
  (set-written e -2))

=>
PITCH: frequency: 233.08186975464196, midi-note: 58, midi-channel: NIL
pitch-bend: 0.0
degree: 116, data-consistent: T, white-note: B3
nearest-chromatic: BF3
src: 0.8908987045288086, src-ref-pitch: C4, score-note: BF3
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: T, natural: NIL,
octave: 3, c5ths: 1, no-8ve: BF, no-8ve-no-acc: B
show-accidental: T, white-degree: 27,
accidental: F,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: BF3, tag: NIL,
data: BF3

;; Create a single-pitch event object, set it's written pitch to two half-steps ;
;; lower, and print the corresponding data slots ;
(let ((e (make-event 'c4 'q)))
  (set-written e -2)
  (print (data (pitch-or-chord e))))
```

```
  (print (data (written-pitch-or-chord e)))))
```

```
=>
C4
BF3
```

SYNOPSIS:

```
(defmethod set-written ((e event) transposition)
```

28.5.153   event/setf amplitude

[event] [Methods]

DESCRIPTION:

Change the amplitude slot of a given event object and automatically add a
mark to set a corresponding dynamic.

Numbers greater than 1.0 and less than 0.0 will also be stored in the
amplitude slot of the given event object without printing a warning, though
corresponding dynamic marks are only available for values between 0.0 and
1.0. Any value above 1.0 or below 0.0 will result in a dynamic marking of
FFFF and NIENTE respectively.

ARGUMENTS:

```
 - An amplitude value (real number).
 - An event object.
```

RETURN VALUE:

 Returns the specified amplitude value.

EXAMPLE:

```
;; When no amplitude is specified, new event objects are created with a default
;; amplitude of 0.7.
(let ((e (make-event 'c4 'q)))
  (amplitude e))

=> 0.7

;; Setting an amplitude returns the amplitude set
(let ((e (make-event 'c4 'q)))
```

```
  (setf (amplitude e) .3))

=> 0.3

;; Create an event object, set its amplitude, then print the contents of the
;; amplitude and marks slots to see the dynamic setting.
(let ((e (make-event 'c4 'q)))
  (setf (amplitude e) .3)
  (print (amplitude e))
  (print (marks e)))

=>
0.3
(PP)

;; Setting an amplitude greater than 1.0 or less than 0.0 sets the amplitude
;; correspondingly and sets the dynamic mark to FFFF or NIENTE respectively.
(let ((e1 (make-event 'c4 'q))
      (e2 (make-event 'c4 'q)))
  (setf (amplitude e1) 1.3)
  (setf (amplitude e2) -1.3)
  (print (marks e1))
  (print (marks e2)))

=>
(FFFF)
(NIENTE)
```

SYNOPSIS:

```
(defmethod (setf amplitude) :after (value (e event))
```

28.5.154   event/setf tempo-change

[event] [Methods]

DESCRIPTION:

 Store the tempo when a change is made.

 NB: This creates a full tempo object, not just a number representing
 bpm. It also sets the display-tempo object to T as a side effect.

ARGUMENTS:

 - An event object.

- A number indicating the new tempo bpm.

RETURN VALUE:

Returns a tempo object.

EXAMPLE:

```
;; Creation of a new event object sets the tempo-change slot to NIL by default,
;; unless otherwise specified.
(let ((e (make-event 'c4 'q)))
  (tempo-change e))

=> NIL

;; The tempo-change method returns a tempo object
(let ((e (make-event 'c4 'q)))
  (setf (tempo-change e) 132))

=>
TEMPO: bpm: 132, beat: 4, beat-value: 4.0, qtr-dur: 0.45454545454545453
       qtr-bpm: 132.0, usecs: 454545, description: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: 132

;; The new tempo object is stored in the event object's tempo-change slot.
(let ((e (make-event 'c4 'q)))
  (setf (tempo-change e) 132)
  e)

=>
EVENT: start-time: NIL, end-time: NIL,
       duration-in-tempo: 0.0,
       compound-duration-in-tempo: 0.0,
       amplitude: 0.7, score-marks: NIL,
       bar-num: -1, cmn-objects-before: NIL,
       tempo-change:
TEMPO: bpm: 132, beat: 4, beat-value: 4.0, qtr-dur: 0.45454545454545453
       qtr-bpm: 132.0, usecs: 454545, description: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: 132
[...]
```

SYNOPSIS:

```
(defmethod (setf tempo-change) (value (e event))
```

28.5.155   event/sharp-p

[event] [Methods]

DESCRIPTION:

 Determine whether the pitch of a given event object has a sharp.

ARGUMENTS:

 - An event object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the test is to handle the written or
   sounding pitch in the event. T = written. Default = NIL.

RETURN VALUE:

 Returns T if the note tested has a sharp, otherwise NIL (ie, is natural or
 has a flat).

EXAMPLE:

```
;; Returns T when the note is sharp
(let ((e (make-event 'cs4 'q)))
  (sharp-p e))

=> T

;; Returns NIL when the note is not sharp (ie, is flat or natural)
(let ((e (make-event 'c4 'q)))
  (sharp-p e))

=> NIL

(let ((e (make-event 'df4 'q)))
  (sharp-p e))

=> NIL
```

SYNOPSIS:

```
(defmethod sharp-p ((e event) &optional written)
```

28.5.156   event/sharp-to-flat

[event] [Methods]

DATE:

 Mon 10 Feb 2020 18:31:17 GMT - London

DESCRIPTION:

 Change the sharp (or sharps) in an event to a flat (or flats).

ARGUMENTS:

 An event object

OPTIONAL ARGUMENTS:

 None

RETURN VALUE:

 An event object.

EXAMPLE:

```
(let ((e (make-event 'as4 'q)))
      (print-simple (sharp-to-flat e)))

=> BF4 Q,
NIL

(let ((e (make-event '(bf4 as4) 'q)))
      (print-simple (sharp-to-flat e)))

=> (BF4 BF4) Q,
NIL

;;; NB This method is non-destructove
(let ((e (make-event 'as4 'q)))
      (print-simple (sharp-to-flat e))
      (print-simple e))

=> BF4 Q,
AS4 Q,
NIL
```

SYNOPSIS:

```
(defmethod sharp-to-flat ((e event) &optional clone written)
```

28.5.157   event/single-pitch-chord-to-pitch

[event] [Methods]

DATE:

```
 Thu 14 Nov 2019 07:44:02 GMT London
```

DESCRIPTION:

 Turn a chord object with only one pitch in its chord slot into a pitch
 object.

ARGUMENTS:

 An event object

OPTIONAL ARGUMENTS:

 NIL

RETURN VALUE:

 An event object containing a new pitch object, if the original chord
 contained a single pitch, or the original event object.

EXAMPLE:

```
(data (single-pitch-chord-to-pitch (make-event '(a4) 'e)))
=> A4
```

SYNOPSIS:

```
(defmethod single-pitch-chord-to-pitch ((e event))
```

28.5.158   event/sort-event-list

[event] [Functions]

DESCRIPTION:

Sort a list of event objects by start time from lowest to highest.

ARGUMENTS:

 - A list of event objects.

RETURN VALUE:

 A list of event objects.

EXAMPLE:

```
;; Create a list of event object with non-sequential start-times, sort them,
;; and return the pitches and start times of the newly ordered list.
(let ((e-list (loop repeat 8
                 for nn in '(c4 d4 e4 f4 g4 a4 b4 c5)
                 for st in '(1.0 3.0 2.0 5.0 8.0 4.0 7.0 6.0)
                 collect (make-event nn 'e :start-time st))))
  (sort-event-list e-list)
  (loop for e in e-list
     collect (get-pitch-symbol e)
     collect (start-time e)))
=>
(C4 1.0 E4 2.0 D4 3.0 A4 4.0 F4 5.0 C5 6.0 B4 7.0 G4 8.0)
```

SYNOPSIS:

```
(defun sort-event-list (event-list)
```

28.5.159   event/transpose

[event] [Methods]

DESCRIPTION:

 Transpose the pitch content of a given event object by a specified number
 of semitones. This method can be applied to chords or single-pitches.

 If functions are given, these will be used for the note or chord in the
 event, whereby semitones may or may not be NIL in that case (transposition
 could be dependent on the note or chord rather than being a fixed shift).

 NB: By default this method returns a modified clone of the original rather
     than changing the values of the original itself. The user can choose to
     replace the values of the original by setting the keyword argument
     :destructively to T.

ARGUMENTS:

- An event object.
- A number (can be positive or negative).

OPTIONAL ARGUMENTS:

keyword arguments:
- :destructively. T or NIL to indicate whether the method is to replace the
  pitch values of the original event object (T) or return a new event
  object with the new pitches (NIL). Default = NIL.
- :chord-function. A function to be used for the transposition of
  chords. Default = #'transpose.
- :pitch-function. A function to be used for the transposition of
  pitches. Default = #'transpose.

RETURN VALUE:

An event object.

EXAMPLE:

```
;; Transpose returns an event object    ;
(let ((e (make-event 'c4 'q)))
  (transpose e 1))

=>
EVENT: start-time: NIL, end-time: NIL,
duration-in-tempo: 0.0,
[...]

;; By default transpose returns a modified clone, leaving the original event ;
;; object untouched.                     ;
(let ((e (make-event 'c4 'q)))
  (print (data (pitch-or-chord (transpose e 1))))
  (print (data (pitch-or-chord e))))

=>
CS4
C4

;; When the keyword argument :destructively is set to T, the data of the ;
;; original event object is replaced    ;
(let ((e (make-event 'c4 'q)))
  (transpose e 1 :destructively t)
```

```
  (data (pitch-or-chord e)))

=> CS4

;; Can transpose by 0 as well (effectively no transposition) ;
(let ((e (make-event 'c4 'q)))
  (transpose e 0 :destructively t)
  (data (pitch-or-chord e)))

=> C4

;; ...or by negative intervals            ;
(let ((e (make-event 'c4 'q)))
  (transpose e -3 :destructively t)
  (data (pitch-or-chord e)))

=> A3

;; Can transpose chords too             ;
(let ((e (make-event '(c4 e4 g4) 'q)))
  (transpose e -3 :destructively t)
  (loop for p in (data (pitch-or-chord e)) collect (data p)))

=> (A3 CS4 E4)
```

SYNOPSIS:

```
(defmethod transpose ((e event) semitones
                      &key
                      destructively
                      ;; the default functions are the class methods for pitch
                      ;; or chord.
                      (chord-function #'transpose)
                      (pitch-function #'transpose))
```

28.5.160   event/voice-chord

[event] [Methods]

DATE:

 November 28th 2018, Heidhausen

DESCRIPTION:

 Set the amplitudes of the individual pitches in a chord event. Amplitude

values should either be integers 0-127 or floating-point 0.0-1.0.

Usually, the amplitude of each pitch in a chord is set by the overall
amplitude slot of the event. However, if the pitch's amplitude slot is a
number, it will be used instead, automatically, when generating MIDI files
(and only those at present).

ARGUMENTS:

 - the event object
 - either a list or a function: if a list, it will be cycled through
   (returning to the beginning if necessary) to set the amplitudes of the
   pitches of the chord from bottom to top; if a function then the function
   is passed the event object and it should return a list of amplitudes which
   will then be used to set the pitch objects' amplitudes similarly.

OPTIONAL ARGUMENTS:

 None

RETURN VALUE:

 The (modified) event object

EXAMPLE:

```
(let ((e (make-event '(c4 d4 e4 f4 g5 a4 b4) 4 :start-time 0.0)))
  (print (voice-chord e '(50 60 70))) ; velocities
  (voice-chord e #'(lambda (e)
                     (loop for p in (get-pitch-symbols e) collect
                         (case p ; amplitude floats
                           (c4 .1) (d4 .2) (e4 .3) (f4 .4) (t .5))))))
```

SYNOPSIS:

```
(defmethod voice-chord ((e event) processor)
```


28.5.161   event/wrap-events-list

[event] [Functions]

DESCRIPTION:

 Given a list of time-ascending event objects, rotate their start-times by

moving the lowest start time to a specified point in the list (determined
either by time or by nth position), assigning the subsequent start times
sequentially through the remainder of events in the list, and wrapping
around to the head of the list again to assign the final remaining start
times. If the first event doesn't start at 0, its start time will be
conserved.

ARGUMENTS:

 - A flat list of event objects.
 - An integer that is the number of the event object with which to start
   (nth position), or a decimal time in seconds.

OPTIONAL ARGUMENTS:

 keyword argument:
 - :time. T or NIL to indicate whether the second argument is a time in
   seconds or an nth index. If a time in seconds, the method skips to the
   closest event object in the list. T = time in seconds. Default = NIL.

RETURN VALUE:

 Returns a flat list of event objects with adjust start-times.

EXAMPLE:

```
;;; Create a list of events of eighth-note durations, specifying start-times at ;
;;; 0.5-second intervals and print the pitches and start-times. Then apply the ;
;;; function and print the pitches and start-times again to see the change. ;
(let ((e-list (loop for st from 1.0 by 0.5
                  for nn in '(c4 d4 e4 f4 g4 a4 b4 c5)
                  collect (make-event nn 'e :start-time st))))
  (print
   (loop for e in e-list
      collect (get-pitch-symbol e)
      collect (start-time e)))
  (wrap-events-list e-list 3)
  (print
   (loop for e in e-list
      collect (get-pitch-symbol e)
      collect (start-time e))))
=>
(C4 1.0 D4 1.5 E4 2.0 F4 2.5 G4 3.0 A4 3.5 B4 4.0 C5 4.5)
(C4 3.5 D4 4.0 E4 4.5 F4 1.0 G4 1.5 A4 2.0 B4 2.5 C5 3.0)
```

SYNOPSIS:

```
(defun wrap-events-list (events start-at &key (time nil))
```

28.5.162   rhythm/force-rest

[rhythm] [Methods]

DESCRIPTION:

 Force the given rhythm object to be a rest.

ARGUMENTS:

 - A rhythm object.

RETURN VALUE:

 A rhythm object.

EXAMPLE:

```
(let ((r (make-rhythm 8)))
  (force-rest r)
  (is-rest r))
```

=> T

SYNOPSIS:

```
(defmethod force-rest ((r rhythm))
```

28.5.163   rhythm/has-mark

[rhythm] [Methods]

DESCRIPTION:

 Check to see if a given rhythm object possesses a specified mark.

ARGUMENTS:

 - A rhythm object.
 - A mark.

RETURN VALUE:

If the specified mark is indeed found in the MARKS slot of the given rhythm object, the tail of the list of marks contained in that slot is returned; otherwise NIL is returned.

EXAMPLE:

```
;; Add a specific mark and check to see if the rhythm object has it.
(let ((r (make-rhythm 'q)))
  (add-mark r 'a)
  (has-mark r 'a))

=> (A)

;; Check to see if the given rhythm object possess a mark we know it doesn't.
(let ((r (make-rhythm 'q)))
  (add-mark r 'a)
  (has-mark r 's))

=> NIL
```

SYNOPSIS:

```
(defmethod has-mark ((r rhythm) mark &optional (test #'equal))
```

28.5.164   rhythm/in-tie

[rhythm] [Methods]

DATE:

 September 14th 2018, Heidhausen

DESCRIPTION:

 return T or NIL depending on whether the rhythm/event is part of a tie
 (either tied-from or tied-to)

ARGUMENTS:

 the rhythm/event object

RETURN VALUE:

 T or NIL

SYNOPSIS:

```
(defmethod in-tie ((r rhythm))
```

28.5.165   rhythm/is-multiple

[rhythm] [Methods]

DESCRIPTION:

Determines if the value of one rhythm object is a multiple of the value of
a second rhythm object. This is established by dividing the one by the
other and checking to see if the quotient is a whole number.

ARGUMENTS:

 - A first rhythm object.
 - A second rhythm object.

RETURN VALUE:

 Returns T if true and NIL if not. Always also returns the quotient.

EXAMPLE:

```
(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'e)))
  (is-multiple r1 r2))

=> T, 2.0

(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'e.)))
  (is-multiple r1 r2))

=> NIL, 1.3333333333333333
```

SYNOPSIS:

```
(defmethod is-multiple ((r1 rhythm) (r2 rhythm))
```

28.5.166   rhythm/make-rhythm

[rhythm] [Functions]

DESCRIPTION:

 Make a rhythm object.

ARGUMENTS:

  - A duration either as a numeric representation of a rhythm (subdivision of
    a whole note; 2 = half note, 4 = quarter, 8 = eighth etc), a quoted
    alphabetic shorthand for a duration (ie, 'h, 'q, 'e etc.), or an absolute
    duration in seconds.

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :is-rest. T or NIL to denote whether the given duration is a rest or
   not. T = rest. Default = NIL.
 - :is-tied-to. T or NIL to denote whether the given duration is tied later
   to the next duration in a given rthm-seq-bar/rthm-seq object. T =
   tied. Default = NIL.
 - :duration. Indicates whether the duration argument has been given as a
   duration in seconds, not a known rhythm like 'e or 8. T indicates that
   the duration is a duration in seconds. Default = NIL.
 - :tempo. Indicates the tempo for the given rhythm and is used only when
    :duration is set to figure out the rhythm type (1/8, 1/4 etc.) from the
    two values. So this is not related to any tempi applied, rather one that
    is reflected in the duration-in-tempo slot of event.
```

RETURN VALUE:

```
 A rhythm object.
```

EXAMPLE:

```
(make-rhythm 16)

=>
RHYTHM: value: 16.0, duration: 0.25, rq: 1/4, is-rest: NIL, score-rthm: 16.0,
        undotted-value: 16, num-flags: 2, num-dots: 0, is-tied-to: NIL,
        is-tied-from: NIL, compound-duration: 0.25, is-grace-note: NIL,
        needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
        rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 16,
        tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 16, tag: NIL,
data: 16

(make-rhythm 16 :is-rest t :is-tied-to t)

=>
RHYTHM: value: 16.0, duration: 0.25, rq: 1/4, is-rest: T, score-rthm: 16.0,
        undotted-value: 16, num-flags: 2, num-dots: 0, is-tied-to: T,
```

```
          is-tied-from: NIL, compound-duration: 0.25, is-grace-note: NIL,
          needs-new-note: NIL, beam: NIL, bracket: NIL, rqq-note: NIL,
          rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 16,
          tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 16, tag: NIL,
data: 16

(make-rhythm .23 :duration t)

=>
RHYTHM: value: 17.391304, duration: 0.23, rq: 23/100, is-rest: NIL, score-rthm: NIL,
          undotted-value: -1.0, num-flags: 0, num-dots: 0, is-tied-to: NIL,
          is-tied-from: NIL, compound-duration: 0.23, is-grace-note: NIL,
          needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
          rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: -1,
          tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: NIL
```

SYNOPSIS:

```
(defun make-rhythm (rthm &key is-rest is-tied-to is-tied-from duration
                             (tempo 60.0)))
```

28.5.167   rhythm/mid-tie

[rhythm] [Methods]

DATE:

```
 September 14th 2018, Heidhausen
```

```
 DESCRIPTION return T or NIL depending on whether the rhythm/event is in the
 middle of a tie (i.e. both tied-from or tied-to are T)
```

ARGUMENTS:

```
 the rhythm/event object
```

RETURN VALUE:

```
 T or NIL
```

SYNOPSIS:

```
(defmethod mid-tie ((r rhythm))
```

28.5.168   rhythm/replace-mark

[rhythm] [Methods]
DESCRIPTION:

 Replace a specified mark of a given rhythm object with a second specified
 mark. If a rhythm object contains more than one mark, individual marks can
 be changed without modifying the remaining marks.

ARGUMENTS:

 - A rhythm object.
 - The mark to be replaced.
 - The new mark.

RETURN VALUE:

 Returns the new value of the MARKS slot of the given object.

EXAMPLE:

```
;; Make a rhythm object, add the mark 'a, then replace 'a with 's
(let ((r (make-rhythm 'q)))
  (add-mark r 'a)
  (replace-mark r 'a 's))

=> (S)

;; Make a rhythm object, add a list of marks, replace just the 'pizz mark with
;; a 'batt mark

(let ((r (make-rhythm 'q)))
  (loop for m in '(a s pizz col-legno) do (add-mark-once r m))
  (replace-mark r 'pizz 'batt))

=> (COL-LEGNO BATT S A)
```

SYNOPSIS:

```
(defmethod replace-mark ((r rhythm) what with &optional ignore)
```

28.5.169   rhythm/rhythm-equal

[rhythm] [Methods]

DESCRIPTION:

 Compares the values of two rhythm objects to determine if they are equal.

 NB rhythm-equal compares the values only, so rhythms with the same values
    will still be considered equal even if their other attributes (such as
    :is-rest and :is-tied-to etc.) are different.

ARGUMENTS:

 - A first rhythm object.
 - A second rhythm object.

RETURN VALUE:

 T if the values of the given rhythm objects are equal, else NIL.

EXAMPLE:

```
(let ((r1 (make-rhythm 4))
      (r2 (make-rhythm 4)))
  (rhythm-equal r1 r2))

=> T

(let ((r1 (make-rhythm 4))
      (r2 (make-rhythm 8)))
  (rhythm-equal r1 r2))

=> NIL

(let ((r1 (make-rhythm 4 :is-rest T))
      (r2 (make-rhythm 4 :is-rest NIL)))
  (rhythm-equal r1 r2))

=> T

(let ((r1 (make-rhythm 4 :is-tied-to T))
      (r2 (make-rhythm 4 :is-tied-to NIL)))
  (rhythm-equal r1 r2))

=> T
```

SYNOPSIS:

```
(defmethod rhythm-equal ((r1 rhythm) (r2 rhythm))
```

28.5.170   rhythm/rhythm-list

[rhythm] [Functions]

DESCRIPTION:

 Create a list of rhythms from symbols, possibly involving ties and not
 needing meters etc. (i.e. not as strict as rthm-seq).

ARGUMENTS:

 - The list of rhythm symbols.

OPTIONAL ARGUMENTS:

 - T or NIL indicates whether to create a circular-sclist from the
   result. If NIL, a simple list will be returned (default = NIL).

RETURN VALUE:

 A list or circular-sclist of the rhythm objects.

EXAMPLE:

```
;; Create a list of rhythm objects
(rhythm-list '(q w+e q. h.+s e.+ts))

=>(
RHYTHM: value: 4.0f0, duration: 1.0
[...]
RHYTHM: value: 1.0f0, duration: 4.0
[...]
RHYTHM: value: 8.0f0, duration: 0.5
[...]
RHYTHM: value: 2.6666666666666665, duration: 1.5
[...]
RHYTHM: value: 1.3333333333333333, duration: 3.0
[...]
RHYTHM: value: 16.0f0, duration: 0.25
[...]
RHYTHM: value: 5.333333333333333, duration: 0.75
```

```
[...]
RHYTHM: value: 24.0f0, duration: 0.16666666666666666
)
```

```
;; Collect the data from each of the individual rhythm objects in the list.
(let ((rl (rhythm-list '(q w+e q. h.+s e.+ts))))
  (print (loop for r in rl collect (data r))))
```

```
=> (Q "W" "E" Q. "H." "S" "E." "TS")
```

```
;; Set the optional argument to T to create a circular-sclist instead
(rhythm-list '(q w+e q. h.+s e.+ts) t)
```

```
=>
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 8, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
[...]
)
```

```
;; Create a circular-sclist and check that it's a circular-sclist using cscl-p
(let ((rl (rhythm-list '(q w+e q. h.+s e.+ts) t)))
  (cscl-p rl))
```

```
=> T
```

SYNOPSIS:

```
(defun rhythm-list (rthms &optional circular)
```

28.5.171   rhythm/rhythm/

[rhythm] [Methods]

DESCRIPTION:

Determines the ratio of one rhythm object's duration to that of a second
rhythm object by use of division.

ARGUMENTS:

- A rhythm object.
- A second rhythm object.

RETURN VALUE:

 A number.

EXAMPLE:

```
(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'e)))
  (rhythm/ r1 r2))

=> 2.0

(let ((r1 (make-rhythm 'q))
      (r3 (make-rhythm 's.)))
  (rhythm/ r1 r3))

=> 2.6666667
```

SYNOPSIS:

```
(defmethod rhythm/ ((r1 rhythm) (r2 rhythm))
```

28.5.172   rhythm/rm-marks

[rhythm] [Methods]

DESCRIPTION:

 Remove a specified mark (or a list of specified marks) from the MARKS slot
 of a given rhythm object. If the mark specified is not present in the given
 rhythm object's MARKS slot, a warning is printed. If some marks of a list
 of specified marks are present in the rhythm object's MARKS slot and other
 aren't, those that are will be removed and a warning will be printed for
 the rest.

ARGUMENTS:

 - A rhythm object.
 - A mark or list of marks.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether a warning is to be printed if the specified
   mark is not present in the given rhythm object's MARKS slot.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; The method itself returns NIL
(let ((r (make-rhythm 'q)))
  (add-mark-once r 'a)
  (rm-marks r 'a))

=> NIL

;; Adding a list of marks to r, then removing only 's
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s pizz col-legno x-head) do
       (add-mark-once r m))
  (rm-marks r 's)
  (marks r))

=> (X-HEAD COL-LEGNO PIZZ A)

;; Removing a list of marks from r
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s pizz col-legno x-head) do
       (add-mark-once r m))
  (rm-marks r '(s a))
  (marks r))

=> (X-HEAD COL-LEGNO PIZZ)

;; Attempting to remove a mark that isn't present results in a warning
;; being printed by default
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s pizz col-legno x-head) do
       (add-mark-once r m))
  (rm-marks r 'zippy))

=> NIL
WARNING: rhythm::rm-marks: no mark ZIPPY in (X-HEAD COL-LEGNO PIZZ S A)

;; Suppress printing the warning when the specified mark isn't present
(let ((r (make-rhythm 'q)))
  (loop for m in '(a s pizz col-legno x-head) do
       (add-mark-once r m))
  (rm-marks r 'zippy nil))
```

```
=> NIL
```

SYNOPSIS:

```
(defmethod rm-marks ((r rhythm) marks &optional (warn t))
```

28.5.173   rhythm/scale

[rhythm] [Methods]

DESCRIPTION:

Change the value of a rhythm object's duration value by a specified
scaling factor. Bear in mind that after scaling, depending on the
complexity of the rhythm, the data slot may no longer accurately represent
the new rhythmic value (though of course the value, duration, and rq, etc.,
slots will).

ARGUMENTS:

- A rhythm object.
- A scaling factor.

OPTIONAL ARGUMENTS:

- <clone>. This argument determines whether a new rhythm object is made or
  the duration value of the old object is replaced. When set to T, a new
  object is made based on the duration value of the original. When set to
  NIL, the original duration value is replaced (see example). Default = T.

RETURN VALUE:

A rhythm object.

EXAMPLE:

```
(let ((r (make-rhythm 4)))
  (data (scale r 2)))
=> H

(let ((r (make-rhythm 4)))
  (data (scale r 3)))
=> H.
```

```
(let ((r (make-rhythm 4)))
  (data (scale r .5)))
=> E
```

SYNOPSIS:

```
(defmethod scale ((r rhythm) scaler &optional (clone t) ignore1 ignore2)
```

28.5.174   rhythm/scale-as-rests

[rhythm] [Methods]

DATE:

 September 29th 2020, Heidhausen

DESCRIPTION:

 Sometimes you don't want to actually scale a rhythm/event's duration rather
 stretch it out with rests. This method will leave the rhythm object intact
 (or cloned) before then cloning it a given number of times but as a rest.

ARGUMENTS:

 - the rhythm object
 - the scaler (integer)

OPTIONAL ARGUMENTS:

 - T or NIL: whether to clone the first argument

RETURN VALUE:

 a list of rhythm objects

SYNOPSIS:

```
(defmethod scale-as-rests ((r rhythm) scaler &optional (clone t))
```

28.5.175   rhythm/start-tie

[rhythm] [Methods]

DATE:

September 14th 2018, Heidhausen

DESCRIPTION:

return T or NIL depending on whether the rhythm/event is the start of a tie
(tied-from but not tied-to)

ARGUMENTS:

the rhythm/event object

RETURN VALUE:

T or NIL

SYNOPSIS:

(defmethod start-tie ((r rhythm))

28.5.176   rhythm/subtract

[rhythm] [Methods]

DESCRIPTION:

Create a new rhythm object with a duration that is equal to the difference
between the duration of two other given rhythm objects.

NB: This method only returns a single rhythm rather than a list with
    ties. Thus h - e., for example, returns TQ...

If the resulting duration cannot be presented as a single rhythm, the DATA
slot of the resulting rhythm object is set to NIL, though the VALUE and
DURATION slots are still set with the corresponding numeric values.

If the resulting duration is equal to or less than 0, NIL is returned and
an optional warning may be printed.

ARGUMENTS:

- A first rhythm object.
- A second rhythm object.

OPTIONAL ARGUMENTS:

    – T or NIL to indicate whether a warning is to be printed when the
      resulting duration is less than or equal to 0. Default = 0.

RETURN VALUE:

 A rhythm object if the resulting duration is greater than 0, else NIL and
 the optional warning.

EXAMPLE:

```
;; Make a new rhythm object with a duration equal to one quarter minus one
;; eighth.
(let ((r1 (make-rhythm 'q))
      (r2 (make-rhythm 'e)))
  (subtract r1 r2))

=>
RHYTHM: value: 8.0f0, duration: 0.5, rq: 1/2, is-rest: NIL, score-rthm: 8.0f0,
        undotted-value: 8, num-flags: 1, num-dots: 0, is-tied-to: NIL,
        is-tied-from: NIL, compound-duration: 0.5, is-grace-note: NIL,
        needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
        rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 8,
        tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E

;; A half minus a dotted eighth is represented as a triplet half
(let ((r1 (make-rhythm 'h))
      (r2 (make-rhythm 'e.)))
  (data (subtract r1 r2)))

=> TQ...

;; If the resulting duration is 0 or less, return NIL, with no warning by
;; default
(let ((r1 (make-rhythm 'e))
      (r2 (make-rhythm 'q)))
  (subtract r1 r2))

=> NIL

;; Setting the optional argument to t returns a warning when the resulting
;; duration is less than 0
(let ((r1 (make-rhythm 'e))
      (r2 (make-rhythm 'q)))
```

```
  (subtract r1 r2 t))

=> NIL
WARNING: rhythm::arithmetic: new duration is -0.5; can't create rhythm

;; Subtracting a septuplet-16th from a quarter results in a duration that
;; cannot be represented as a single rhythm, therefore setting the DATA to NIL
;; while VALUE and DURATION are still set.
(let ((r1 (make-rhythm 4))
      (r2 (make-rhythm 28)))
  (print (value (subtract r1 r2)))
  (print (duration (subtract r1 r2)))
  (print (data (subtract r1 r2))))

=>
4.666666666666666
0.8571428571428572
NIL
```

SYNOPSIS:

```
(defmethod subtract ((r1 rhythm) (r2 rhythm) &optional warn)
```

28.5.177   linked-named-object/sclist

[linked-named-object] [Classes]

NAME:

```
 player

 File:             sclist.lsp

 Class Hierarchy:  named-object -> linked-named-object -> sclist

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of a simple but self-checking (hence
                   sclist) list class.

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    February 11th 2001
```

```
$$ Last modified:  14:20:05 Sat Jan  4 2020 CET
```

```
SVN ID: $Id$
```

28.5.178   sclist/change-data

[sclist] [Classes]

NAME:

```
change-data
```

```
File:             change-data.lsp
```

```
Class Hierarchy:  named-object -> linked-named-object -> sclist ->
                  change-data
```

```
Version:          1.0.11
```

```
Project:          slippery chicken (algorithmic composition)
```

```
Purpose:          Implementation of the change-data class.  Holds data
                  regarding parameter changes for a whole section
                  (e.g. tempo).  For use in change-map.  The data in the
                  <changes> slot is a three-element list: the sequence
                  number, the bar number of the sequence where the change
                  takes place (defaults to 1) and the new data (e.g. a
                  tempo value).

                  When giving this data, the sequence number and bar
                  numbers are always integers > 0, unlike sequences
                  themselves which may be given any kind of id.  Therefore
                  it's OK to sort the given data according to integer
                  precedence and perform numeric tests on them too.

                  No public interface envisaged (so no robodoc entries).
```

```
Author:           Michael Edwards: m@michael-edwards.org
```

```
Creation date:    2nd April 2001
```

```
$$ Last modified: 20:31:51 Mon May 14 2012 BST
```

```
SVN ID: $Id$
```

28.5.179   change-data/get-change-data

[change-data] [Methods]

DESCRIPTION:

 Get the change data (for example, from an instrument-change-map object) for
 a specified sequence.

ARGUMENTS:

 - A change-data object.
 - An integer that is the number of the sequence within the given
   change-data object for which to retrieve the data.

OPTIONAL ARGUMENTS:

 - An integer that is the number of the bar within the specified sequence
   for which to return the change data.

RETURN VALUE:

 The change data of the specified sequence (and bar).

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                     (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s)
                                 (w))
                                :pitch-seq-palette ((1 2 3 4 5 6)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))))))
  (get-change-data
   (get-data '(1 sax) (instrument-change-map mini)) 2 2))

=> ALTO-SAX, NIL
```

SYNOPSIS:

```
(defmethod get-change-data ((cd change-data) sequence &optional (bar 1))
```

28.5.180 change-data/make-change-data

[change-data] [Functions]

DESCRIPTION:

Create a change-data object, which holds data for use by a change-map
object. The data stored in change-data object will be that of parameter
changes for a whole section, such as tempo values.

The data is passed to the make-change-data function as a list of
three-element lists, each consisting of the number of the sequence, the
number of the bar within that sequence, and the new data.

ARGUMENTS:

- An ID for the change-data object to be created.
- A list of three-item lists, each consisting of the number of the sequence
  in which the data is to change, the number of the bar within that
  sequence in which the data is to change, and the data value itself. The
  sequence number and bar number are always integers > 0. If no bar-number
  is given, it will default to 1.

RETURN VALUE:

A change-data object.

EXAMPLE:

```
(make-change-data 'cd-test '((1 1 23) (6 1 28) (18 1 35)))

=>
CHANGE-DATA:
            previous-data: NIL,
            last-data: 35
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: CD-TEST, tag: NIL,
data: ((1 1 23) (6 1 28) (18 1 35))
```

SYNOPSIS:

```
(defun make-change-data (id data)
```

28.5.181   sclist/chord

[sclist] [Classes]

NAME:

```
chord

File:            chord.lsp

Class Hierarchy:  named-object -> linked-named-object -> sclist -> chord

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Implementation of the chord class that is simply an
                  sclist whose data is a list of pitch instances.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    July 28th 2001

$$ Last modified:  11:20:52 Tue Aug  4 2020 CEST

SVN ID: $Id$
```

28.5.182   chord/add-harmonics

[chord] [Methods]

DESCRIPTION:

```
Adds pitches to the set which are harmonically related to the existing
pitches. The keywords are the same as for the get-harmonics function.  See
also sc-set method of the same name and get-pitch-list-harmonics. NB This
will automatically sort all pitches from high to low.
```

ARGUMENTS:

```
- a chord object
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
 see get-harmonics function
```

RETURN VALUE:

 the same set object as the first argument but with new pitches added.

SYNOPSIS:

(defmethod add-harmonics ((c chord) &rest keywords)

28.5.183   chord/add-mark

[chord] [Methods]

DESCRIPTION:

 Add the specified mark to the MARKS slot of the given chord object.

ARGUMENTS:

 - A chord object.
 - A mark.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print a warning when attempting to add a
   mark to a rest.

RETURN VALUE:

 Returns the full contents of the MARKS slot of the given chord object

EXAMPLE:

```
;;; Returns the complete contents of the MARKS slot
(let ((chrd (make-chord '(cs4 e4 fs4 af4 bf4))))
  (add-mark chrd 'fff)
  (add-mark chrd 'pizz))

=> (PIZZ FFF)
```

SYNOPSIS:

(defmethod add-mark ((c chord) mark &optional warn-rest warn-again)

28.5.184 chord/add-pitches

[chord] [Methods]

DESCRIPTION:

Add the specified pitches to the given chord object. This can include
duplicate pitches for things like violin fingered and open string
unisons. If you don't want pitch duplications, call the rm-duplicates
method after calling this method.

ARGUMENTS:

- A chord object.
- The pitches to add to that object. These can be pitch objects or any data
  that can be passed to make-pitch, or indeed lists of these, as they will
  be flattened.

RETURN VALUE:

- A chord object.

EXAMPLE:

```
(let ((ch (make-chord '(c4 e4 g4))))
  (print (get-pitch-symbols ch))
  (add-pitches ch 'bf3)
  (print (get-pitch-symbols ch))
  (add-pitches ch 'af3 'a4 'b4)
  (print (get-pitch-symbols ch))
  (add-pitches ch '(cs5 ds5 fs5))
  (print (get-pitch-symbols ch)))

=>
(C4 E4 G4)
(BF3 C4 E4 G4)
(AF3 BF3 C4 E4 G4 A4 B4)
(AF3 BF3 C4 E4 G4 A4 B4 CS5 DS5 FS5)
```

SYNOPSIS:

```
(defmethod add-pitches ((c chord) &rest pitches)
```

28.5.185 chord/artificial-harmonic-simple?

[chord] [Methods]

DATE:

 January 10th 2019, Heidhausen

DESCRIPTION:

 Tests a chord to see if it's an artificial harmonic such as that created by
 force-artificial-harmonic. Unlike artificial-harmonic? this looks for
 two-note chords, the second of which has the flag-head marks and is a fourth
 away i.e. this is the simplest double-octave type.

ARGUMENTS:

 - a chord object

RETURN VALUE:

 T or NIL

SYNOPSIS:

(defmethod artificial-harmonic-simple? ((c chord))

28.5.186   chord/artificial-harmonic?

[chord] [Methods]

DATE:

 October 4th 2014

DESCRIPTION:

 Determine whether a chord represents an artificial harmonic of the type
 that strings play. An artificial harmonic here is defined as a three note
 chord where the second note has a 'flag-head mark (i.e diamond shape) and
 the 1st and 3rd are related in frequency by an integer ratio.

 NB What we don't do (yet) is test whether the 2nd note is the correct nodal
 point to produce the given pitch.

ARGUMENTS:

 - a chord object

OPTIONAL ARGUMENTS:

 - cents-tolerance: how many cents the top note can deviate from a pure
   partial frequency. E.g. the 7th harmonic is about 31 cents from the
   nearest tempered note.

RETURN VALUE:

 If the chord is an artificial harmonic then the sounding (3rd) note is
 returned, otherwise NIL.

SYNOPSIS:

(defmethod artificial-harmonic? ((c chord) &optional (cents-tolerance 31))

28.5.187   chord/calculate-dissonance

[chord] [Methods]

DATE:

 27th July 2015, Edinburgh

DESCRIPTION:

 Calculates the dissonance of a chord as it would appear when played on the
 piano (by default; other instruments' spectral data can also be used: see
 below). This uses a roughness calculation model by Pantelis N. Vassilakis
 as outlined at http://www.acousticslab.org/learnmoresra/moremodel.html. The
 model has been perceptually verified. We use spectral data handled by
 spectra.lsp to sum the roughness of sine pairs up to the first 12 partials
 of each tone, i.e. every partial of every tone is calculated in relation to
 all other tones' partials, taking their amplitudes into account.

 NB If notes are above/below the piano range for which we have data then we
 use the highest/lowest spectral data available.

 NB This method will be called automatically the first time you access a
 chord object's dissonance slot. In that case the default keyword arguments
 will be used. If you want to use different arguments or recalculate
 dissonance after the chord has been altered, this method can be used at any
 time but bear in mind that it does automatically change the dissonance
 slot so use with caution. If you merely want to change the
 default spectra which are used, then you can do that at startup via
 (set-sc-config 'default-spectra ...). You can also call rm-diss-cen to

delete centroid and dissonance slot values (this exists in the set-palette
class too).

ARGUMENTS:

- the chord object

OPTIONAL ARGUMENTS:

keyword arguments:
- :num-partials. The number of partials we want to use in our
  calculation. Default = 12.
- :average. T or NIL to indicate whether we want to use the average
  spectrum for an octave with the current note of a chord in the middle.
  Default = T.
- :spectrum. Pass a two-element list of partial frequency scalers (list)
  and amplitudes (list) to use instead of the piano data. The amplitudes
  should be normalised from 0.0 to 1.0 and there should be as many of these
  as frequency scalers and :num-partials. Bear in mind that even using the
  same data for several calls, this will still give different results for
  the same chord type starting on different notes, as the perceptual
  dissonance is based on pitch height as well as interval structure.
  Default = (get-sc-config 'default-spectra) (see globals.lsp)

RETURN VALUE:

A floating point number representing the dissonance value. Higher values
are more dissonant.

EXAMPLE:

```
(loop for chord in '((c4 e4 g4)
                     (d4 fs4 a4)
                     (c4 ef4 g4)
                     (c2 e2 g2)
                     (c3 e3 g3)
                     (c4 e4 g4 b4)
                     (c4 e4 g4 b4 cs4))
   collect (calculate-dissonance (make-chord chord) :num-partials 12))
-->
(0.402377778595695d0 0.3699311905936456d0 0.4174836004632471d0
 1.8766497834038562d0 1.0522737914732592d0 0.7389679913974341d0
 1.59662137873394d0)
```

SYNOPSIS:

```
(defmethod calculate-dissonance ((c chord)
                                 &key (num-partials 12) (average t)
                                   (spectrum (get-sc-config 'default-spectra)))
```

28.5.188   chord/calculate-spectral-centroid

[chord] [Methods]

DATE:

 July 28th 2015

DESCRIPTION:

 Calculate the spectral centroid of a chord. This technique is of course
 usually applied in digital signal processing to an audio signal via a
 Fast Fourier Transform (FFT). It's perhaps a little strange to use this on
 pitch data, not least of which because we will not be taking into
 account any phase information or interference of possible harmonic partial
 interactions, but nevertheless we get a good indication of the "pitch
 height" of a chord via this method.

 NB This will be called automatically the first time you access a chord
 object's centroid slot. In that case the default keyword arguments will be
 used. If you want to use different arguments or recalculate the centroid
 after the chord has been altered, this method can be used at any time but
 bear in mind that it does automatically change the centroid slot so use
 with caution. You can call rm-diss-cen to delete centroid and dissonance
 slot values (this exists in the set-palette class too).

ARGUMENTS:

 - the chord object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :num-partials. The number of partials we want to use in our
   calculation. Default = 12.
 - :average. T or NIL to indicate whether we want to use the average
   spectrum for an octave with the current note of a chord in the middle.
   Default = T.
 - :spectrum. Pass a two-element list of partial frequency scalers (list)
   and amplitudes (list) to use instead of the piano data. The amplitudes
   should be normalised from 0.0 to 1.0 and there should be as many of these

as frequency scalers and :num-partials. Default = (get-sc-config
'default-spectra) (see globals.lsp)

RETURN VALUE:

A floating point value representing the frequency in Hertz of the spectral
centroid.

EXAMPLE:

```
(calculate-spectral-centroid (make-chord '(fs3 c4 cs4 d4 ds4 e4)))
 --> 692.3215865354373d0
(calculate-spectral-centroid (make-chord '(fs3 c4 cs4 d4 ds4 e4 g6)))
 --> 772.4924013974714d0
```

SYNOPSIS:

```
(defmethod calculate-spectral-centroid ((c chord)
                                        &key (num-partials 12)
                                          (average t)
                                          (spectrum
                                           (get-sc-config 'default-spectra)))
```

28.5.189   chord/chord-equal

[chord] [Methods]

DESCRIPTION:

Test to see if two chords are equal.

NB: Two unsorted chord objects that contain the exact same pitch objects in
    a different order will not be considered equal and will return NIL.

NB: Equality is tested on pitch content only, not on, for example, the
    values of the MIDI slots of those pitch objects etc.

ARGUMENTS:

- A first chord object.
- A second chord object.

RETURN VALUE:

T or NIL. T if the pitch content of the chords is equal, otherwise NIL.

EXAMPLE:

```
;; Two chords are equal
(let ((chrd1 (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                         :midi-channel 11
                         :microtones-midi-channel 12))
      (chrd2 (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                         :midi-channel 11
                         :microtones-midi-channel 12)))
  (chord-equal chrd1 chrd2))

=> T

;; Chord objects with the same pitch objects in a different order are unequal
(let ((chrd1 (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                         :midi-channel 11
                         :microtones-midi-channel 12))
      (chrd2 (make-chord '(e4 c4 gqs4 bqf4 d5 f5)
                         :midi-channel 11
                         :microtones-midi-channel 12
                         :auto-sort nil)))
  (chord-equal chrd1 chrd2))

=> NIL

;; Only the pitch content is compared. Content of other slots is irrelevant.
(let ((chrd1 (make-chord '(e4 c4 gqs4 bqf4 d5 f5)
                         :midi-channel 11
                         :microtones-midi-channel 12))
      (chrd2 (make-chord '(e4 c4 gqs4 bqf4 d5 f5)
                         :midi-channel 7
                         :microtones-midi-channel 8)))
  (chord-equal chrd1 chrd2))

=> T
```

SYNOPSIS:

```
(defmethod chord-equal ((c1 chord) (c2 chord))
```

28.5.190   chord/chord-member

[chord] [Methods]

DESCRIPTION:

Test whether a specified pitch object is a member of a given chord object.

ARGUMENTS:

- A chord object.
- A pitch object. This must be a pitch object, not just a note-name symbol,
  but the pitch object can be made with either a note-name symbol or a
  numerical hertz frequency value.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether or not the method should consider
  enharmonically equivalent pitches to be equal. T = enharmonics are
  equal. Default = T.

RETURN VALUE:

Similar to Lisp's "member" function, this method returns the tail of the
data (list of pitch objects) of the tested chord object starting with the
specified pitch object if that pitch is indeed a member of that list,
otherwise returns NIL.

NB: Since the method returns the tail of the given chord (the "rest" of the
    pitches after the given pitch), the result may be different depending
    on whether that chord has been auto-sorted or not.

EXAMPLE:

```
;; Returns the tail of pitch objects contained starting with the tested pitch
(let ((chrd (make-chord '(c4 e4 gqs4 a4 d5 f5 bqf5)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'a4))))

=> (A4 D5 F5 BQF5)

;; The chord object's default auto-sort feature might appear to affect outcome
(let ((chrd (make-chord '(d5 c4 gqs4 a4 bqf5 f5 e4)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'a4))))

=> (A4 D5 F5 BQF5)

;; Returns NIL if the pitch is not present in the tested chord object. This
```

```
;; example uses the "pitch-list-to-symbols" function to simplify the
;; pitch-object output.
(let ((chrd (make-chord '(d5 c4 gqs4 a4 bqf5 f5 e4)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'b4))))

=> NIL

;; The optional <enharmonics-are-equal> argument is set to NIL by default
(let ((chrd (make-chord '(c4 e4 a4 d5 f5))))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'ds4))))

=> NIL

;; Setting the optional <enharmonics-are-equal> argument to T
(let ((chrd (make-chord '(c4 ef4 a4 d5 f5))))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'ds4) t)))

=> (EF4 A4 D5 F5)

;; The optional <octaves-are-true> argument is NIL by default

(let ((chrd (make-chord '(c4 ef4 a4 d5 ef5 f5))))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'c5))))

=> NIL

;; If optional <octaves-are-true> argument is set to T, any occurrence of the
;; same pitch class in a different octave will be considered part of the chord
;; and return a positive result.
(let ((chrd (make-chord '(c4 ef4 a4 d5 ef5 f5))))
  (pitch-list-to-symbols (chord-member chrd (make-pitch 'c5) nil t)))

=> (C4 EF4 A4 D5 EF5 F5)
```

SYNOPSIS:

```
(defmethod chord-member ((c chord) (p pitch)
                         &optional (enharmonics-are-equal t)
                                   (octaves-are-true nil))
```

28.5.191   chord/chord=

[chord] [Methods]

DATE:

 November 11th 2013

DESCRIPTION:

 Test whether the pitch objects of the two chords are pitch=. Assumes both
 chords are sorted by pitch height. See pitch= in the pitch class for
 details of comparing pitches.

RETURN VALUE:

 T or NIL

SYNOPSIS:

```
(defmethod chord= ((c1 chord) (c2 chord) &optional enharmonics-are-equal
                   (frequency-tolerance 0.01)) ; (src-tolerance 0.0001))
```

28.5.192   chord/collapse

[chord] [Methods]

DATE:

 February 1st 2018, Heidhausen

DESCRIPTION:

 Collapse or shift all the pitches of a chord until a single octave.

ARGUMENTS:

 - a chord object
 - the octave to shift the pitches into (integer)

OPTIONAL ARGUMENTS:

RETURN VALUE:

 a new chord object with the pitches in the given object

EXAMPLE:

```
(print-simple (collapse (make-chord '(d4 f4 bf4 e5 b5)) 2))
--> NIL: D2 E2 F2 BF2 B2
```

SYNOPSIS:

```
(defmethod collapse ((c chord) octave)
```

28.5.193   chord/common-notes

[chord] [Methods]

DESCRIPTION:

Return the integer number of pitches common to two chord objects.

ARGUMENTS:

- A first chord object.
- A second chord object.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether enharmonically equivalent pitches are to be
  considered the same pitch. T = enharmonically equivalent pitches are
  considered the same pitch. Default = T.
- T or NIL to indicate whether the same pitch class in different octaves is
  to be considered the same pitch. T = consider the same pitch class from
  octaves to be the same pitch. Default = NIL.

RETURN VALUE:

Returns an integer that is the number of pitches common to the two chords
objects. Two further values are returned: the list of common pitches and a
list of their pitch symbols.

EXAMPLE:

```
;; The following two chord objects have 3 pitches in common
(let ((chrd-1 (make-chord '(c4 e4 g4 b4 d5 f5)))
      (chrd-2 (make-chord '(d3 f3 a3 c4 e4 g4))))
  (common-notes chrd-1 chrd-2))
=> 3
```

```
;; By default, enharmonically equivalent pitches are considered to be the same
;; pitch
(let ((chrd-1 (make-chord '(c4 e4 g4 b4 d5 f5)))
      (chrd-2 (make-chord '(d3 f3 a3 c4 ff4 g4))))
  (common-notes chrd-1 chrd-2))
=> 3

;; Setting the first optional argument to NIL causes enharmonically equivalent
;; pitches to be considered separate pitches
(let ((chrd-1 (make-chord '(c4 e4 g4 b4 d5 f5)))
      (chrd-2 (make-chord '(d3 f3 a3 c4 ff4 g4))))
  (common-notes chrd-1 chrd-2 nil))
=> 2

;; By default, the same pitch class in different octaves is considered to be a
;; separate pitch
(let ((chrd-1 (make-chord '(c4 e4 g4 b4 d5 f5)))
      (chrd-2 (make-chord '(d3 f3 a3 ff4 g4 c5))))
  (common-notes chrd-1 chrd-2 t))
=> 2

;; Setting the second optional argument to T causes all pitches of the same
;; pitch class to be considered equal regardless of their octave
(let ((chrd-1 (make-chord '(c4 e4 g4 b4 d5 f5)))
      (chrd-2 (make-chord '(d3 f3 a3 ff4 g4 c5))))
  (common-notes chrd-1 chrd-2 t t))
=> 5
```

SYNOPSIS:

```
(defmethod common-notes ((c1 chord) (c2 chord)
                         &optional (enharmonics-are-equal t)
                         (octaves-are-true nil))
```

28.5.194   chord/contains-pitches

[chord] [Methods]

DESCRIPTION:

Check to see if a given chord object contains pitch objects for all of the
specified note-names. The method returns NIL if any one of the specified
pitches is not found in the given chord object.

ARGUMENTS:

- An chord object.
- A list of note-name symbols or pitch objects. NB: If checking for only one
  pitch, that pitch must be passed as a single-item list.

RETURN VALUE:

 T or NIL.

EXAMPLE:

```
;; Returns T when all specified pitches are contained in the given chord
;; object
(let ((c (make-chord '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (contains-pitches c '(d2 e3 gf4 af5)))

=> T

;; Returns NIL if any one of the specified pitches is not contained in the
;; given chord object.
(let ((c (make-chord '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (contains-pitches c '(d2 e3 gf4 b4 af5)))

=> NIL
```

SYNOPSIS:

```
(defmethod contains-pitches ((c chord) pitches
                                 &optional enharmonics-are-equal octaves-are-true)
```

28.5.195   chord/delete-marks

[chord] [Methods]

DESCRIPTION:

 Delete all marks from the MARKS slot of the given chord object.

ARGUMENTS:

 - A chord object.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
;;; Make a chord object, add two marks, and print the MARKS slot to see them;
;;; apply delete-marks and print the MARKS slot again to see the change
(let ((chrd (make-chord '(cs4 e4 fs4 af4 bf4))))
  (add-mark chrd 'fff)
  (add-mark chrd 'pizz)
  (print (marks chrd))
  (delete-marks chrd)
  (print (marks chrd)))

=>
(PIZZ FFF)
NIL
```

SYNOPSIS:

```
(defmethod delete-marks ((c chord))
```

28.5.196   chord/get-highest

[chord] [Methods]

DATE:

 28/10/14

DESCRIPTION:

 Assuming the auto-sort slot is T (which it is by default), return the last
 (= highest) pitch in the chord.

ARGUMENTS:

 - a chord object

RETURN VALUE:

 The highest pitch (object) in the chord.

SYNOPSIS:

```
(defmethod get-highest ((c chord))
```

28.5.197 chord/get-interval-structure

[chord] [Methods]

DESCRIPTION:

Get the distances between each pitch in a given sc-set object and the lowest (or neighbouring) pitch in that object in DEGREES (which default to quarter-tones in slippery chicken). This method assumes that the given sc-set object is sorted from low to high, which is the default action for sc-set objects.

ARGUMENTS:

- An sc-set object.

OPTIONAL
- T or NIL indicating whether to return values in semitones or default of degrees. Special case: if this argument is 'frequencies, then the interval structure will be returned as frequency differences. T = semitones. Default = NIL.
- T or NIL to indicated whether we should return intervals from pitch to pitch (T) or from the lowest to each pitch (NIL). Default = NIL.

RETURN VALUE:

A list of integers.

EXAMPLE:

```
;;; Returns the distances in degrees (which are quarter-tones by default
;;; in slippery chicken--use (in-scale :chromatic) at the top of your code to
;;; set to the chromatic scale):

(get-interval-structure (make-chord '(c4 e4 g4)))

=> (8.0 14.0)

;;; Return semitones
(get-interval-structure (make-chord '(c4 e4 g4)) t))

;;; Interval structure not from lowest but from pitch to pitch (ascending)
(get-interval-structure (make-chord '(c4 e4 g4 b4)) t t)
(4.0 3.0 4.0)

=> (4.0 7.0)
```

SYNOPSIS:

```
(defmethod get-interval-structure ((c chord) &optional
                                           in-semitones neighbour)
```

28.5.198   chord/get-midi-channel

[chord] [Methods]

DESCRIPTION:

Get the MIDI channel of the first pitch object contained in a given chord
object.

NB: This method returns only the midi-channel of the first pitch object in
    the chord object's data list.

ARGUMENTS:

 - A chord object.

RETURN VALUE:

 An integer: the midi channel

EXAMPLE:

```
(let ((chrd (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (get-midi-channel chrd))

=> 11
```

SYNOPSIS:

```
(defmethod get-midi-channel ((c chord))
```

28.5.199   chord/get-partials-amps

[chord] [Methods]

DATE:

 July 28th 2015, Edinburgh

DESCRIPTION:

Return a list containing for every pitch in the chord a list of two lists:
first the partial frequency scalars (which will be integers or very close
to integers) then the normalised partial amplitudes.

Note that if spectrum are given then average is ignored. If spectrum is an
assoc-list then we get the data from it otherwise if it's just a list of
freq-scalers then amps, it's fixed and we use that repeatedly.

ARGUMENTS:

- the chord object

OPTIONAL ARGUMENTS:

keyword arguments:
- :average. Whether to average spectral data over one octave centring around
each of the pictures in the chord. Default = T.
- :spectrum. The spectrum data. Default = (get-sc-config 'default-spectra)
  (see globals.lsp)

RETURN VALUE:

A list of two element sublist. of numbers.

EXAMPLE:

```
(get-partials-amps (make-chord '(c4 e4 g4)))
-->
((((0.99666667 1.9958333 2.9941666 3.9958336 5.002501 6.015 7.0333333 8.049168
   9.086666 10.125 11.174167 12.234166)
  (0.9447856 1.0000001 0.22975834 0.13516104 0.1625952 0.05319352 0.12118169
   2.1978654e-4 0.049031425 0.010756988 0.011732774 0.0))
 ((0.9966666 1.9958333 2.9966667 4.0025005 5.0133333 6.028333 7.0575004
   8.085834 9.136668 10.193334 11.264999 12.346667)
  (0.99999994 0.73724264 0.20146085 0.10752921 0.122650035 0.03613218
   0.06638182 0.0 0.022111647 0.004390597 0.011441806 5.167861e-4))
 ((0.9966666 1.995 3.0 4.005834 5.0208335 6.042501 7.08 8.1225 9.185 10.261666
   11.45 11.405833)
  (1.0 0.48279247 0.1664997 0.066108614 0.07859015 0.027295936 0.035848983 0.0
   0.010107763 0.0027390774 0.0040924386 0.0010194636)))
```

SYNOPSIS:

```
(defmethod get-partials-amps ((c chord)
                              &key (average t)
                                (spectrum (get-sc-config 'default-spectra)))
```

28.5.200   chord/get-pitch

[chord] [Methods]

DESCRIPTION:

Get the pitch object located at the specified index within the given chord
object. The <ref> argument is 1-based.

ARGUMENTS:

- A chord object.
- An integer that is the index of the pitch object sought within the data
  list of the given chord object.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
(let ((chrd (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (get-pitch chrd 3))

=>
PITCH: frequency: 403.482, midi-note: 67, midi-channel: 12
       pitch-bend: 0.5
       degree: 135, data-consistent: T, white-note: G4
       nearest-chromatic: G4
       src: 1.5422108173370361, src-ref-pitch: C4, score-note: GS4
       qtr-sharp: 1, qtr-flat: NIL, qtr-tone: 1,
       micro-tone: T,
       sharp: NIL, flat: NIL, natural: NIL,
       octave: 4, c5ths: 0, no-8ve: GQS, no-8ve-no-acc: G
       show-accidental: T, white-degree: 32,
       accidental: QS,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: GQS4, tag: NIL,
data: GQS4
```

SYNOPSIS:

```
(defmethod get-pitch ((c chord) ref)
```

28.5.201   chord/get-pitch-symbols

[chord] [Methods]

DESCRIPTION:

Return the data of the pitch objects from a given chord object as a list of
note-name symbols.

ARGUMENTS:

- A chord object.

RETURN VALUE:

A list of note-name symbols.

EXAMPLE:

```
(let ((chrd (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                          :midi-channel 11
                          :microtones-midi-channel 12)))
  (get-pitch-symbols chrd))

=> (C4 E4 GQS4 BQF4 D5 F5)
```

SYNOPSIS:

```
(defmethod get-pitch-symbols ((c chord) &optional ignore)
```

28.5.202   chord/has-notes

[chord] [Methods]

DATE:

16-Aug-2010

DESCRIPTION:

Tests whether a given chord object contains at least one pitch
object.

(make-chord nil) is a valid function call and creates a chord object with
no notes.

ARGUMENTS:

- A chord object.

RETURN VALUE:

Returns T if the given chord object contains at least one pitch object,
otherwise returns NIL.

EXAMPLE:

```
;; Returns T if the given chord object contains at least one pitch object
(let ((chrd (make-chord '(c4))))
  (has-notes chrd))

=> T

(let ((chrd (make-chord '(c4 e4 g4))))
  (has-notes chrd))

=> T

;; Otherwise returns NIL
(let ((chrd (make-chord nil)))
  (has-notes chrd))

=> NIL
```

SYNOPSIS:

```
(defmethod has-notes ((c chord))
```

28.5.203   chord/highest

[chord] [Methods]

DESCRIPTION:

Return the pitch object from the given chord object that has the highest
pitch data.

NB: As opposed to the "lowest" method, this method cannot handle chord
    objects whose pitches have not been auto-sorted from low to high.

ARGUMENTS:

- A chord object.

RETURN VALUE:

 A pitch object

EXAMPLE:

```
;; Returns the last pitch object of a chord object
(let ((chrd (make-chord '(e4 c4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (highest chrd))

=>
PITCH: frequency: 698.456, midi-note: 77, midi-channel: 11
       pitch-bend: 0.0
       degree: 154, data-consistent: T, white-note: F5
       nearest-chromatic: F5
       src: 2.669679641723633, src-ref-pitch: C4, score-note: F5
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 5, c5ths: 0, no-8ve: F, no-8ve-no-acc: F
       show-accidental: T, white-degree: 38,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: F5, tag: NIL,
data: F5

;; Is not capable of returning the highest pitch object from chord objects that
;; have not been auto-sorted
(let ((chrd (make-chord '(e4 c4 gqs4 bqf4 f5 d5)
                        :midi-channel 11
                        :microtones-midi-channel 12
                        :auto-sort nil)))
  (data (highest chrd)))

=> D5
```

SYNOPSIS:

```
(defmethod highest ((c chord))
```

28.5.204   chord/invert

[chord] [Methods]

DATE:

 May 1st 2020, Heidhaisen

DESCRIPTION:

 Invert the interval structure of a chord. By default the given bottom note
 becomes the new top note, as the intervals are mirrored around it, but if
 the optional argument is non-NIL then the result will be transposed so that
 the chord has the original range.

ARGUMENTS:

 - a chord object

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether the retain the original range or not

RETURN VALUE:

 a new chord object which is the interval inversion of the argument

EXAMPLE:

```
(get-pitch-symbols (invert (make-chord '(d4 f4 bf4 e5 b5))))
--> (F2 C3 FS3 B3 D4)

(get-pitch-symbols (invert (make-chord '(d4 f4 bf4 e5 b5)) t))
--> (D4 A4 EF5 AF5 B5)
```

SYNOPSIS:

```
(defmethod invert ((c chord) &optional top-to-bottom ignore)
```

28.5.205   chord/lowest

[chord] [Methods]

DESCRIPTION:

Return the pitch object from the given chord object that has the lowest
pitch data. The method can handle chord objects whose pitches have not been
auto-sorted from low to high.

ARGUMENTS:

- A chord object.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
;; Returns the pitch object of the lowest pitch despite not being sorted
(let ((chrd (make-chord '(e4 c4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
                        :microtones-midi-channel 12
                        :auto-sort nil)))
  (lowest chrd))

=>
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 11
       pitch-bend: 0.0
       degree: 120, data-consistent: T, white-note: C4
       nearest-chromatic: C4
       src: 1.0, src-ref-pitch: C4, score-note: C4
       qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
       micro-tone: NIL,
       sharp: NIL, flat: NIL, natural: T,
       octave: 4, c5ths: 0, no-8ve: C, no-8ve-no-acc: C
       show-accidental: T, white-degree: 28,
       accidental: N,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4
```

SYNOPSIS:

```
(defmethod lowest ((c chord))
```

28.5.206   chord/make-chord

[chord] [Functions]

DESCRIPTION:

 Create a chord object from a list of note-name symbols.

ARGUMENTS:

 - A list of note-name symbols.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :id. An element of any type that is to be the ID of the chord object
   created.
 - :auto-sort. T or NIL to indicate whether the method should first sort the
   individual pitch objects created from low to high before returning the
   new chord object. T = sort. Default = T.
 - :midi-channel. An integer that is to be the MIDI channel value to which
   all of the chromatic pitch objects in the given chord object are to be
   set for playback. Default = 1.
 - :microtones-midi-channel. An integer that is to be the MIDI channel value
   to which all of the microtonal pitch objects in the given chord object
   are to be set for playback. Default = 1. NB: See
   player.lsp/make-player for details on microtones in MIDI output.
 - :force-midi-channel. T or NIL to indicate whether to force a given value
   to the MIDI-CHANNEL slot, even if the notes passed to the method are
   already pitch objects with non-zero MIDI-CHANNEL values.

RETURN VALUE:

 A chord object.

EXAMPLE:

```
;; Simple usage with default values for keyword arguments
(make-chord '(c4 e4 g4 b4 d5 f5))

=>
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL
SCLIST: sclist-length: 6, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
```

```
PITCH: frequency: 329.628, midi-note: 64, midi-channel: 0
[...]
data: E4
[...]
PITCH: frequency: 391.995, midi-note: 67, midi-channel: 0
[...]
data: G4
[...]
PITCH: frequency: 493.883, midi-note: 71, midi-channel: 0
[...]
data: B4
[...]
PITCH: frequency: 587.330, midi-note: 74, midi-channel: 0
[...]
data: D5
[...]
PITCH: frequency: 698.456, midi-note: 77, midi-channel: 0
[...]
data: F5
)


;; By default the pitches are first sorted low to high
(let ((mc (make-chord '(e4 c4 g4 b4 f5 d5))))
  (loop for p in (data mc) collect (data p)))

=> (C4 E4 G4 B4 D5 F5)


;; Setting the :midi-channel and :microtones-midi-channel arguments results in
;; the MIDI-CHANNEL slot of each of the contained pitch objects being set
;; accordingly, depending on whether it is a chromatic or microtonal pitch
(let ((mc (make-chord '(cqs4 e4 gqf4 b4 dqf5 f5)
                      :midi-channel 11
                      :microtones-midi-channel 12)))
  (loop for p in (data mc) collect (midi-channel p)))

=> (12 11 12 11 12 11)
```

SYNOPSIS:

```
(defun make-chord (note-list &key (id nil) (auto-sort t) (midi-channel 1)
                   (microtones-midi-channel 1) (force-midi-channel t))
```

28.5.207   chord/make-chord-from-intervals

[chord] [Functions]

DATE:

 May 1st 2020, Heidhausen

DESCRIPTION:

 Make a chord object from a starting pitch and a list of intervals in
 semitones.

ARGUMENTS:

 - the starting pitch: symbol or pitch object
 - the list of intervals in semitones (may be fractional if current scale is
 microtonal)

OPTIONAL ARGUMENTS:

 any keyword argument accepted by make-chord

RETURN VALUE:

 a chord object

EXAMPLE:

```
(get-pitch-symbols (make-chord-from-intervals 'c4 '(6 5 7.5) :midi-channel 7))
--> (C4 FS4 B4 GQF5)
```

SYNOPSIS:

```
(defun make-chord-from-intervals (start-pitch intervals
                                  &rest keyargs &key &allow-other-keys)
```

28.5.208   chord/morph

[chord] [Methods]

DATE:

 May 3rd 2016, Edinburgh

DESCRIPTION:

 Attempt to morph between two chords by selecting some notes from each
 according to the amount given.

ARGUMENTS:

```
- the first chord object
- the second chord object
- the amount to morph from chord 1 to 2. This should be a number between 0
and 1 where 0 would mean all chord 1, 1 would mean all chord 2 and anything
else is an actual morph.
```

RETURN VALUE:

```
A new chord object representing a mixture of pitches from the two
chords. The length of this will depend on the relative lengths of the two
chords but the minimum will be the length of the smallest chord and the
maximum the length of the largest chord (in terms of number of pitches).

NB If you call this method on an sc-set object the subsets and related-sets
slots will be NIL (i.e. not copied over/morphed from either of the two
arguments).
```

EXAMPLE:

```
(print-simple
 (morph (make-chord '(c4 e4 g4 b4)) (make-chord '(df4 f4 af4 c5)) 0.5))
--> C4 F4 G4 C5
```

SYNOPSIS:

```
(defmethod morph ((c1 chord) (c2 chord) amount)
```

28.5.209   chord/no-accidental

[chord] [Methods]

DESCRIPTION:

```
Set the SHOW-ACCIDENTAL slot of all pitch objects within a given chord
object to NIL. This results in no accidentals for the given chord being printed
when written to a score, and also excludes the writing of any accidentals
for that chord in parentheses.
```

ARGUMENTS:

```
- A chord object.
```

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;;; Make a chord, print the SHOW-ACCIDENTAL slots of the pitch objects it
;;; contains; then call the method and print the same slots again to see the
;;; change.

(let ((chrd (make-chord '(cs4 e4 fs4 af4 bf4))))
  (print (loop for p in (data chrd) collect (show-accidental p)))
  (no-accidental chrd)
  (print (loop for p in (data chrd) collect (show-accidental p))))

=>
(T T T T T)
(NIL NIL NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod no-accidental ((c chord))
```

28.5.210   chord/num-notes

[chord] [Methods]

DATE:

 June 22nd 2019, Heidhausen

DESCRIPTION:

 A helper function to return the number of notes in a chord (which is merely
 the sclist-length slot as a chord is a child class of sclist)

ARGUMENTS:

 - the chord object

RETURN VALUE:

 an integer

SYNOPSIS:

```
(defmethod num-notes ((c chord))
```

28.5.211 chord/output-midi-note

[chord] [Methods]

DESCRIPTION:

Generate the MIDI-related data for each pitch in a given chord object.

ARGUMENTS:

- A chord object.
- A number that is the start time in seconds of the given chord within the
  output MIDI file.
- A decimal number between 0.0 and 1.0 that is the amplitude of the given
  chord in the output MIDI file.
- A number that is the duration in seconds of the given chord in the output
  MIDI file.

RETURN VALUE:

The corresponding data in list form.

EXAMPLE:

```
;; Generate the MIDI-related data required for a 5-note chord that starts 100
;; seconds into the output MIDI file, with an amplitude of 0.5 and a duration
;; of 13.0 seconds.
(let ((chrd (make-chord '(cs4 e4 fs4 af4 bf4))))
  (output-midi-note chrd 100.0 0.5 13.0))

=> (#i(midi time 100.0 keynum 61 duration 13.0 amplitude 0.5 channel -1)
    #i(midi time 100.0 keynum 64 duration 13.0 amplitude 0.5 channel -1)
    #i(midi time 100.0 keynum 66 duration 13.0 amplitude 0.5 channel -1)
    #i(midi time 100.0 keynum 68 duration 13.0 amplitude 0.5 channel -1)
    #i(midi time 100.0 keynum 70 duration 13.0 amplitude 0.5 channel -1))
```

SYNOPSIS:

```
(defmethod output-midi-note ((c chord) time amplitude duration)
```

28.5.212 chord/pitch-

[chord] [Methods]

DESCRIPTION:

Determine the difference between the lowest pitch of two chords. This
method can be used, for example, to compare the written and sounding
versions of a chord to determine transposition.

If the lower chord is passed as the first argument, the method will return
a negative number.

NB: This method takes pitch bend into consideration when calculating.

ARGUMENTS:

 - A first chord object.
 - A second chord object.

RETURN VALUE:

 A positive or negative decimal number.

EXAMPLE:

```
;; The method measures the distance between the first (lowest) pitches of the
;;; chord only.
(let ((chrd-1 (make-chord '(c4 e4 g4)))
      (chrd-2 (make-chord '(d4 e4 fs4 a4))))
  (pitch- chrd-2 chrd-1))

=> 2.0

;;; Passing the lower chord as the first argument produces a negative result
(let ((chrd-1 (make-chord '(c4 e4 g4)))
      (chrd-2 (make-chord '(d4 e4 fs4 a4))))
  (pitch- chrd-1 chrd-2))

=> -2.0
```

SYNOPSIS:

```
(defmethod pitch- ((c1 chord) (c2 chord))
```

28.5.213   chord/pitch-or-chord=

[chord] [Methods]

DATE:

Wed 18 Sep 2019 17:37:39 BST - London

DESCRIPTION:

Convenience method, test to see if the pitch-or-chord slots of two event
objects are the same.

ARGUMENTS:

- a pitch or chord object
- a pitch or chord object

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether or not enharmonic pitches are considered
  equal. T = enharmonic pitches are considered equal. Default = NIL.
- a number to indicate the frequency deviation allowed before returning NIL.

RETURN VALUE:

T if the values of the two specified pitch or chord objects are equal,
otherwise NIL.

EXAMPLE:

```
;; Comparison of equal pitch objects created using note-name symbols returns T
(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'c4)))
  (pitch-or-chord= p1 p2))

=> T

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2))

=> NIL

(let ((p1 (make-pitch 'c4))
      (p2 (make-pitch 'bs3)))
  (pitch-or-chord= p1 p2 t))

=> T

(let ((p1 (make-pitch 'c4))
```

```
      (c1 (make-chord '(c4 e4 g4))))
  (pitch-or-chord= p1 c1))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(c4 e4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs4 ff4 g4))))
  (pitch-or-chord= c1 c2))

=> NIL

(let ((c1 (make-chord '(c4 e4 g4)))
      (c2 (make-chord '(bs3 ff4 g4))))
  (pitch-or-chord= c1 c2 t))

=> T

;; Chords with only one pitch are the same as pitch object with the same pitch.
(let ((c (make-chord '(c4)))
      (p (make-pitch 'c4)))
  (pitch-or-chord= c p))

=> T
```

SYNOPSIS:

```
(defmethod pitch-or-chord= ((c1 chord) (c2 chord)
                            &optional enharmonics-are-equal
                              (frequency-tolerance 0.01))
```

28.5.214 chord/respell-chord

[chord] [Methods]

DESCRIPTION:

```
 Respell the pitches of a given chord object to improve interval structure;
 i.e., removing augmented intervals etc.
```

This method respells pitches from the bottom of the chord upwards. It does
not process the pitches downwards again once pitches has been
changed. Instead, it reattempts the whole respelling with the enharmonic of
the lowest pitch to determine which spelling produces the fewest
accidentals.

NB: Respelling pitches in a chord is a rather complex process and is by no
    means fool-proof. The process employed here is based on avoiding double
    accidentals; thus, since both FS4 and BS4 have single sharps, the BS4
    won't be changed to C5.

ARGUMENTS:

 - A chord object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print feedback from the process to the
   listener. T = print. Default = NIL.

RETURN VALUE:

 A chord object.

EXAMPLE:

```
(let ((chrd (make-chord '(a3 ds4 f4 fs5 c6))))
  (pitch-list-to-symbols (data (respell-chord chrd t))))

=> (A3 EF4 F4 GF5 C6)
```

SYNOPSIS:

```
(defmethod respell-chord ((c chord) &optional verbose)
```

28.5.215   chord/rm-pitches

[chord] [Methods]

DESCRIPTION:

 Remove the specified pitches from an existing chord object.

ARGUMENTS:

- A chord object.
- The pitches to remove from that object. These can be pitch objects or any
  data that can be passed to make-pitch, or indeed lists of these, as they
  will be flattened. NB: No warning/error will be signalled if the pitches
  to be removed are not actually in the chord.

RETURN VALUE:

- A chord object.

EXAMPLE:

```
(let ((ch (make-chord '(af3 bf3 c4 e4 g4 a4 b4 cs5 ds5 fs5))))
  (print (get-pitch-symbols ch))
  (rm-pitches ch 'bf3)
  (print (get-pitch-symbols ch))
  (rm-pitches ch 'af3 'a4 'b4)
  (print (get-pitch-symbols ch))
  (rm-pitches ch '(cs5 ds5 fs5))
  (print (get-pitch-symbols ch)))

=>
(AF3 BF3 C4 E4 G4 A4 B4 CS5 DS5 FS5)
(AF3 C4 E4 G4 A4 B4 CS5 DS5 FS5)
(C4 E4 G4 CS5 DS5 FS5)
(C4 E4 G4)
```

SYNOPSIS:

```
(defmethod rm-pitches ((c chord) &rest pitches)
```

28.5.216   chord/round-to-nearest

[chord] [Methods]

DATE:

October 18th 2018

DESCRIPTION:

Round the pitches in the chord to the nearest pitches in the current
or given scale. See the pitch class for more details.

ARGUMENTS:

the chord object

OPTIONAL ARGUMENTS:

keyword argument:
:scale. The scale to use when rounding. (Common Music tuning object or
symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or 'quarter-tone
only at present. Default is the current scale as set by (in-scale :...).

RETURN VALUE:

the modified chord object

SYNOPSIS:

(defmethod round-to-nearest ((c chord) &key (scale cm::*scale*))

28.5.217   chord/set-midi-channel

[chord] [Methods]

DESCRIPTION:

Set the MIDI channel of the pitch objects in a given chord object to the
specified values.

ARGUMENTS:

- A chord object.
- An integer that is to be the MIDI channel for chromatic pitches in the
  given chord object.
- An integer that is to be the MIDI channel for microtonal pitches in the
  given chord object. NB: See player.lsp/make-player for details on
  microtones in MIDI output.

RETURN VALUE:

Always returns NIL.

EXAMPLE:

```
;; Returns NIL
(let ((chrd (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
```

```
                               :microtones-midi-channel 12)))
  (set-midi-channel chrd 3 4))

=> NIL

;; Print the value of the MIDI slot for each of the pitch objects contained in
;; the chord object before and after setting
(let ((chrd (make-chord '(c4 e4 gqs4 bqf4 d5 f5)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (print (loop for p in (data chrd) collect (midi-channel p)))
  (set-midi-channel chrd 3 4)
  (print (loop for p in (data chrd) collect (midi-channel p))))

=>
(11 11 12 12 11 11)
(3 3 4 4 3 3)
```

SYNOPSIS:

```
(defmethod set-midi-channel ((c chord) midi-channel
                             &optional microtones-midi-channel)
```

28.5.218   chord/similarity

[chord] [Methods]

DATE:

 12th August 2015, Wals, Austria

DESCRIPTION:

 Calculates the similarity of two chord objects on a scale of 0.0-1.0. This
 weights equally the number of common notes and the interval structure
 between adjacent notes.

ARGUMENTS:

 - the first chord object
 - the second chord object

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether enharmonic notes should be treated as

equal. Even when NIL, enharmonic equivalents will still score high
  overall, just not perfect.
- T or NIL to indicate whether octave equivalents such as f#1 and f#2
  should result in higher scores.

RETURN VALUE:

 a number between 0.0 and 1.0, with 1.0 being the same exact chord and 0.0
 being two chords with no notes and no adjacent intervals in common.

EXAMPLE:

```
(similarity (make-chord '(c4 e4 g4)) (make-chord '(c4 e4 g4)))
=> 1.0
(similarity (make-chord '(c4 e4 g4)) (make-chord '(df4 f4 af4)))
=> 0.5
(similarity (make-chord '(c1 cs1 f1 b1)) (make-chord '(d1 e1 fs1 gs1 as1)))
=> 0.0
(similarity (make-chord '(cs4 es4 gs4)) (make-chord '(df4 f4 af4)))
=> 1.0
(similarity (make-chord '(cs4 es4 gs4)) (make-chord '(df4 f4 af4)) nil)
=> 0.75
```

SYNOPSIS:

```
(defmethod similarity ((c1 chord) (c2 chord)
                       &optional (enharmonics-are-equal t)
                         (octaves-are-true nil))
```

28.5.219   chord/single-pitch-chord-to-pitch

[chord] [Methods]
DATE:

 Thu 14 Nov 2019 07:44:02 GMT London

DESCRIPTION:

 Turn a chord object with only one pitch in its chord slot into a pitch
 object.

ARGUMENTS:

 A chord object

OPTIONAL ARGUMENTS:

 NIL

RETURN VALUE:

 The new pitch object if the original chord contained a single pitch,
 otherwise the original chord object.

EXAMPLE:

```
(data (single-pitch-chord-to-pitch (make-chord '(a4))))
=> A4
```

SYNOPSIS:

```
(defmethod single-pitch-chord-to-pitch ((c chord))
```

28.5.220   chord/sort-pitches

[chord] [Methods]

DESCRIPTION:

 Sort the pitch objects contained within a given chord object and return
 them as a list of pitch objects.

 As an optional argument, 'ascending or 'descending can be given to indicate
 whether to sort from low to high or high to low.

ARGUMENTS:

 - A chord object.

OPTIONAL ARGUMENTS:

 - The symbol 'ASCENDING or 'DESCENDING to indicate whether to sort the
   given pitch objects from low to high or high to low.
   Default = 'ASCENDING.

RETURN VALUE:

 Returns the chord with its pitches sorted.

EXAMPLE:

```
;; Apply the method with no optional argument (defaults to 'ASCENDING) and
;; collect and print the data of the pitch objects in the resulting list
(let ((chrd (make-chord '(d5 c4 gqs4 bqf5 f5 e4)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (print (loop for p in (sort-pitches chrd) collect (data p))))

=> (C4 E4 GQS4 D5 F5 BQF5)

;; Sort from high to low
(let ((chrd (make-chord '(d5 c4 gqs4 bqf5 f5 e4)
                        :midi-channel 11
                        :microtones-midi-channel 12)))
  (print (loop for p in (sort-pitches chrd 'descending) collect (data p))))

=> (BQF5 F5 D5 GQS4 E4 C4)
```

SYNOPSIS:

```
(defmethod sort-pitches ((c chord) &optional (order 'ascending))
```

28.5.221   chord/thin

[chord] [Methods]

DATE:

```
October 25th 2018, Heidhausen
```

DESCRIPTION:

```
Reduce the number of notes in a chord. See the sc-set class method for
details.
```

SYNOPSIS:

```
(defmethod thin ((c chord) &key (strength 5) remove target invert)
```

28.5.222   chord/top-to-bottom

[chord] [Methods]

DATE:

```
May 1st 2020, Heidhausen
```

DESCRIPTION:

 Transpose a chord so that it's current top note is the new bottom note. NB
 This is destructive by default

ARGUMENTS:

 - a chord object

OPTIONAL ARGUMENTS:

 keywords arguments:
 - :destructively. Whether to modify the chord object or return a new
 chord. Default = T

RETURN VALUE:

 the modified chord object

EXAMPLE:

```
(print-simple (top-to-bottom (make-chord '(c4 e4 g4 b4))))
--> NIL: B4 EF5 FS5 BF5
```

SYNOPSIS:

```
(defmethod top-to-bottom ((c chord) &key (destructively t))
```

28.5.223   chord/transpose

[chord] [Methods]

DESCRIPTION:

 Transpose the pitches of a given chord object by a specified number of
 semitones. The specified number can be positive or negative, and may
 contain a decimal segment for microtonal transposition. If passed a decimal
 number, the resulting note-names will be scaled to the nearest degree of
 the current tuning.

ARGUMENTS:

 - A chord object.
 - A positive or negative integer or decimal number indicating the number of
   semitones by which the pitches of the given chord object are to be
   transposed.

OPTIONAL ARGUMENTS:

```
keyword arguments
- :destructively. Whether to change the internal structure of the first
  argument or leave it unchanged, returning a new transposed chord object
  instead. Default = T.
- :do-related-sets. T or NIL to indicate whether to transpose any contents
  of the RELATED-SETS slot as well. T = transpose.  Default = NIL.
- :lowest. Don't transpose pitches in the original chord which are lower
  than  this argument. Default = NIL but eventually via the
  transpose-pitch-list function C-1 (midi note 0)
- :highest. Don't transpose pitches in the original chord which are higher
  than this argument. Default = NIL but eventually B8 (midi note 119)
```

RETURN VALUE:

```
 the transposed chord object
```

EXAMPLE:

```
;; Returns a chord object
(let ((chrd (make-chord '(c4 e4 g4))))
  (transpose chrd 3))

=>
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
[...]
)

;; Accepts positive and negative integers and decimal numbers
(let ((chrd (make-chord '(c4 e4 g4))))
  (pitch-list-to-symbols (data (transpose chrd 3))))

=> (EF4 G4 BF4)

(let ((chrd (make-chord '(c4 e4 g4))))
  (pitch-list-to-symbols (data (transpose chrd -3))))

=> (A3 CS4 E4)

(let ((chrd (make-chord '(c4 e4 g4))))
```

```
  (pitch-list-to-symbols (data (transpose chrd -3.17))))
```

=> (AQF3 CQS4 EQF4)

SYNOPSIS:

```
(defmethod transpose ((c chord) semitones
                      &key (destructively t) ; do-related-sets
                        ;; MDE Tue Aug 21 19:49:47 2018
                        lowest highest)
```

28.5.224   chord/wrap

[chord] [Methods]

DATE:

 March 5th 2016, Edinburgh

DESCRIPTION:

 Take the highest pitch and wrap it around to the bottom, placing it at the
 same interval to the former lowest pitch as the highest pitch is to the
 next-to-highest pitch. Then, by default, we shift (transpose) the chord so
 that the next-to-highest pitch is the former highest pitch.

 NB If the <transpose> optional argument is T, then the lowest and highest
 notes of the chord will always remain the same no matter how many times this
 operation is performed.

ARGUMENTS:

 - A chord object

OPTIONAL ARGUMENTS:

 - the number of times to perform this operation

RETURN VALUE:

 The original chord object with the new pitches (clone in advance if
 necessary)

EXAMPLE:

```
(get-pitch-symbols (wrap (make-chord '(c4 e4 g4 b4))))
=> (C4 E4 GS4 B4)

(get-pitch-symbols (wrap (make-chord '(c4 fs4 b4 f5))))
=> (C4 GF4 C5 F5)

(get-pitch-symbols (wrap (make-chord '(df3 c4 fs4 b4 f5))))
=> (DF3 G3 FS4 C5 F5)

(get-pitch-symbols (wrap (make-chord '(df3 c4 fs4 b4 f5)) 2))
=> (DF3 GF3 C4 B4 F5)

(get-pitch-symbols (wrap (make-chord '(d4 f4 bf4 e5 b5)) 1 nil))
=> (G3 D4 F4 BF4 E5)
```

SYNOPSIS:

```
(defmethod wrap ((c chord) &optional (num-times 1) (transpose t))
```

28.5.225   sclist/circular-sclist

[sclist] [Classes]

NAME:

```
 circular-sclist

 File:            circular-sclist.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         Implementation of the circular-sclist class which offers
                  the use of a function to cycle through the values in the
                  sclist, starting at the beginning again once we've
                  reached the end.

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   February 19th 2001
```

```
$$ Last modified:  11:26:03 Sat Sep 26 2020 CEST

 SVN ID: $Id$
```

28.5.226   circular-sclist/afu

[circular-sclist] [Classes]

NAME:

```
 afu

 File:            afu.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> afu

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         AFU = Alternativ fuer Unentschiedener (a little play on
                  the rather silly German political party).  This translates
                  to Alternative for the Undecided. Building on the
                  circular-sclist and activity-levels class, it's meant as a
                  deterministic alternative to randomness.

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   May 18th 2019

 $$ Last modified:  10:30:25 Tue Aug 11 2020 CEST
```

28.5.227   afu/binlist-to-proportions

[afu] [Functions]

DATE:

```
 May 23rd 2019, Heidhausen
```

DESCRIPTION:

```
 From a list of an arbitrary number of 1s and 0s, create a list of
 corresponding proportions: Each occurrence of 1 starts a new item, with each
```

subsequent occurence of 0 incrementing the proportion we'll collect.

NB all 0s at the beginning of the list will be ignored until we see the
first 1.

ARGUMENTS:

 a simple list of 1s and 0s

RETURN VALUE:

 a list of integers

EXAMPLE:

```
(binlist-to-proportions '(1 0 0 1 0 1 0 1 0 0 0))
--> (3 2 2 4)

;;; the final 1 has no following 0s so its 'length' is 1
(binlist-to-proportions '(1 0 0 1 0 1 0 1 0 0 0 1))
--> (3 2 2 4 1)

;;; all leading zeros ignored
(binlist-to-proportions '(0 0 0 1 0 0 1 0 1 0 1 0 0 0 1))
--> (3 2 2 4 1)
```

SYNOPSIS:

```
(defun binlist-to-proportions (binlist)
```


28.5.228   afu/inflate-proportionally

[afu] [Functions]

DATE:

 May 23rd 2019, Heidhausen

DESCRIPTION:

 Take a list of proportions (generally integers but floating-point works too)
 and create a longer list, using the proportions scaled proportionally. In
 other words a list like '(1 2 3) will become '((1 2 3) (2 4 6) (3 6 9))
 i.e. internally always in proportion like the original but leading of course
 then to other proportions when used with permutations (see inline comments).

ARGUMENTS:

 a simple list of proportions

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :invert. Whether to invert proportions (e.g. 3 becomes 1/3) . Default =
   T.
 - :rescale. Whether to scale the results to be within new
   minima/maxima. This should be a two-element list (min. max.). Default =
   '(0 1).
 - :highest. The highest number to use from the original list when scaling.
   This limit is offered so that we don't end up with most of our numbers down
   at the very bottom end of our scale. Default = 4.
 - :lowest. Similar to :highest but the minimum scaler we'll use. Default = 1.
 - :reverse. Whether to alternate the original order with the reverse thereof
   when looping through the list and applying proportional scalers. Default =
   T.
 - :expt. An exponent to raise our results to before the rescaling process
   (if used). Of course this completely changes proportions but there are
   times when results are far too skewed to the bottom of the range so using
   an exponent such as 0.3

RETURN VALUE: SYNOPSIS:

```
(defun inflate-proportionally (l &key (invert t) (rescale '(0 1))
                                 (lowest 1) (highest 4)
                                 (reverse t) (expt nil))
```

28.5.229   afu/make-afu

[afu] [Functions]

DATE:

 May 23rd 2019, Heidhausen

DESCRIPTION:

 Make an afu object. See https://michael-edwards.org/wp/?p=1227 for examples.

ARGUMENTS:

 Any keyword arguments that make-instance would accept i.e. slots such as
 start-at, level, minimum...

RETURN VALUE:

 An afu object

EXAMPLE:

```
(make-afu :level 3 :minimum -3 :maximum 3 :exponent .3)

AFU: start-at: 0, level: 3, period: 113
     minimum: -3, maximum: 3, num-unique: 32
     unique: (-3.0 -2.908942 -2.8265874 -2.7273216 -2.6483202 -2.5553133
              -2.4432077 -2.3418174 -2.2489507 -2.2196069 -2.1223445 -1.8698215
              -1.7599787 -1.7449956 -1.6306628 -1.5945365 -1.4747927 -1.3338206
              -1.1639004 -1.0286679 -0.86946154 -0.824985 -0.6775627 -0.5040059
              -0.29480958 -0.12831879 0.3039422 0.7751665 0.98014116 1.5123172
              2.3448153 3.0)
     binlist: (1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0
               0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0
               1 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0
               1 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1)
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 408, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (-2.1223445 -1.4747927 -2.5553133 -1.4747927 -0.6775627 -2.4432077
       -2.1223445 -0.6775627 -1.8698215 -2.1223445 -1.4747927 -2.5553133
       -1.4747927 -0.6775627 -2.4432077 -2.1223445 -0.6775627 -1.8698215
       -2.1223445 -1.4747927 -2.5553133 -1.4747927 -0.6775627 -2.4432077
       -2.1223445 -0.6775627 -1.8698215 -2.1223445 -1.4747927 -2.5553133
       -1.4747927 -0.6775627 -2.4432077 -0.6775627 -1.1639004 -2.7273216
       -1.1639004 -1.8698215 -2.8265874 -1.8698215 -2.4432077 -2.2196069
...
```

SYNOPSIS:

```
(defun make-afu (&rest keyargs &key &allow-other-keys)
```

28.5.230   afu/scale

[afu] [Methods]

DATE:

 May 23rd 2019, Heidhausen

DESCRIPTION:

Scale the existing data list to be within new bounds. NB this does not check
the existing data list for its maximum and minimum values, rather it uses
the existing minimum and maximum slot values (which should be correct,
unless something went wrong).

ARGUMENTS:

- the afu object
- the new desired minimum (number). Default = NIL = use the current minimum
  slot.

OPTIONAL ARGUMENTS:

- the new desired maximum (number). Default = NIL = use the current maximum
  slot

RETURN VALUE:

 the new data list

SYNOPSIS:

```
(defmethod scale ((a afu) new-min &optional new-max ignore1 ignore2)
```

28.5.231   circular-sclist/assoc-list

[circular-sclist] [Classes]

NAME:

 assoc-list

 File: assoc-list.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the assoc-list class that is somewhat
                   like the lisp association list but with more

```
                      error-checking.

 Author:              Michael Edwards: m@michael-edwards.org

 Creation date:       February 18th 2001

 $$ Last modified:   15:10:57 Fri Sep 25 2020 CEST

 SVN ID: $Id$
```

28.5.232   assoc-list/add

[assoc-list] [Methods]

DESCRIPTION:

Add a new element to the given assoc-list object. By default the new element
is added at the end of the data list but the optional argument can change
this.

ARGUMENTS:

 - A key/data pair as a list, or a named-object.
 - The assoc-list object to which it is to be added.

OPTIONAL ARGUMENTS:

 - T or NIL to add items to the front of the data list. Default = NIL = to
the end

RETURN VALUE:

Returns T if the specified named-object is successfully added to the given
assoc-list.

Returns an error if an attempt is made to add NIL to the given assoc-list
or if the given named-object is already present in the given assoc-list.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (add '(makers mark) al))
```

```
=> T

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (add '(makers mark) al)
  (get-data 'makers al))

=>
NAMED-OBJECT: id: MAKERS, tag: NIL,
data: MARK

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (add '(makers mark) al)
  (get-position 'makers al))

=> 3

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (add '(knob creek) al))

=> T
```

SYNOPSIS:

```
(defmethod add (named-object (al assoc-list) &optional front)
```

28.5.233   assoc-list/add-to-list-data

[assoc-list] [Methods]

DESCRIPTION:

 Add an element of any type to the end of the data (list) associated with a
 given key of a given assoc-list.

 The data associated with the given key must already be a list.

ARGUMENTS:

 - An item of any type.

- A given key that must be present in the given assoc-list.
- The given assoc-list.

RETURN VALUE:

 Returns the whole named-object to which the new element was added.

 This method will abort with an error if a key is sought which does not
 exist within the given assoc-list. For such cases, use
 add-to-list-data-force instead.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog (fido spot))
                                   (cow bessie)))))
  (add-to-list-data 'rover 'dog al))

=>
NAMED-OBJECT: id: DOG, tag: NIL,
data: (FIDO SPOT ROVER)
```

SYNOPSIS:

```
(defmethod add-to-list-data (new-element key (al assoc-list))
```

28.5.234   assoc-list/add-to-list-data-force

[assoc-list] [Methods]

DESCRIPTION:

 Similar to add-to-list-data, but if the given key doesn't already exist in
 the given assoc-list, it is first added, then the given new element is
 added to that as a 1-element list.

 If the given key already exists within the given assoc-list, its data must
 already be in the form of a list.

ARGUMENTS:

- A (new) element of any type.
- A given key that may or may not be present in the given assoc-list.
- The the given assoc-list.

RETURN VALUE:

 Returns the whole named-object to which the element was added when used
 with a key that already exists within the given assoc-list.

 Returns T when used with a key that does not already exist in the given
 assoc-list.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog (fido spot))
                                   (cow bessie)))))
  (add-to-list-data-force 'rover 'dog al))

=>
NAMED-OBJECT: id: DOG, tag: NIL,
data: (FIDO SPOT ROVER)

(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog (fido spot))
                                   (cow bessie)))))
  (add-to-list-data-force 'wilbur 'pig al)
  (get-keys al))

=> (CAT DOG COW PIG)
```

SYNOPSIS:

```
(defmethod add-to-list-data-force (new-element key (al assoc-list))
```

28.5.235   assoc-list/ascending-ids?

[assoc-list] [Methods]

DATE:

 September 19th 2020, Heidhausen

DESCRIPTION:

 Check whether all IDs are ascending integers (though order can be any), none
 missing, optionally starting from particular integer.

ARGUMENTS:

- the assoc-list object

OPTIONAL ARGUMENTS:

- an integer representing the lowest required ID

RETURN VALUE:

T or NIL

EXAMPLE:

```
(ascending-ids? (make-assoc-list 'test '((1 dog) (2 cat) (3 horse)))) -> T
;;; 3 is missing
(ascending-ids? (make-assoc-list 'test '((1 dog) (2 cat) (4 horse)))) -> NIL
;; doesn't start at 2
(ascending-ids? (make-assoc-list 'test '((1 dog) (2 cat) (3 horse))) 2) -> NIL
;; missing several integers
(ascending-ids? (make-assoc-list 'test '((4 dog) (2 cat) (7 horse))) 2) -> NIL
(ascending-ids? (make-assoc-list 'test '((4 dog) (2 cat) (3 horse))) 2) -> T
```

SYNOPSIS:

```
(defmethod ascending-ids? ((al assoc-list) &optional starting-from)
```

28.5.236   assoc-list/get-data

[assoc-list] [Methods]

DESCRIPTION:

Return the named-object (id, tag and data) that is identified by a
specified key within a given assoc-list.

NB: This method returns the named object itself, not just the data
    associated with the key (use get-data-data for that).

ARGUMENTS:

- A symbol that is the key (id) of the named-object sought.
- The assoc-list object in which it is be sought.

OPTIONAL ARGUMENTS:

     - T or NIL to indicate whether a warning is printed if the specified key
      cannot be found within the given assoc-list. T = print. Default = T.
      Mostly we define whether we want to warn in the instance itself, but
      sometimes it would be good to warn or not on a call basis, hence the
      optional argument.

RETURN VALUE:

 A named-object is returned if the specified key is found within the given
 assoc-list object.

 NIL is returned and a warning is printed if the specified key is not found
 in the given assoc-list object.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-data 'four al))

=>
NAMED-OBJECT: id: FOUR, tag: NIL,
data: ROSES

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-data 'jack al))

=> NIL
WARNING:
   assoc-list::get-data: Could not find data with key JACK in assoc-list with
   id TEST

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-data 'jack al t))

=> NIL
WARNING:
   assoc-list::get-data: Could not find data with key JACK in assoc-list with
   id TEST

(let ((al (make-assoc-list 'test '((jim beam)
```

```
                                        (four roses)
                                        (wild turkey)))))
  (get-data 'jack al nil))
```

=> NIL

SYNOPSIS:

```
(defmethod get-data (key (al assoc-list) &optional (warn t))
```

28.5.237 assoc-list/get-data-data

[assoc-list] [Methods]

DESCRIPTION:

```
 (Short-cut for (data (get-data ...))
 Get the data associated with the given key of the given assoc-list.
```

ARGUMENTS:

```
 - The assoc-list key symbol associated with the data list which is sought.
 - The assoc-list in which it is to be sought.
```

OPTIONAL ARGUMENTS:

```
 - T or NIL to indicate whether to print a warning if no such named-object
   can be found within the given assoc-list (default = T).
```

RETURN VALUE:

```
 If the given key is found within the given assoc-list, the data associated
 with that key is returned.
```

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-data-data 'jim al))
```

=> BEAM

```
(let ((al (make-assoc-list 'test '((jim beam)
```

```
                                      (four roses)
                                      (wild turkey)))))
  (get-data-data 'jack al))
```

```
=> NIL
WARNING:
   assoc-list::get-data: Could not find data with key JACK in assoc-list with
   id TEST
```

SYNOPSIS:

```
(defmethod get-data-data (key (al assoc-list) &optional (warn t))
```

28.5.238   assoc-list/get-first

[assoc-list] [Methods]

DESCRIPTION:

 Returns the first named-object in the DATA slot of the given assoc-list
 object.

ARGUMENTS:

 - An assoc-list object.

RETURN VALUE:

 A named-object that is the first object in the DATA slot of the given
 assoc-list object.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-first al))
```

```
=>
NAMED-OBJECT: id: JIM, tag: NIL,
data BEAM
```

SYNOPSIS:

```
(defmethod get-first ((al assoc-list))
```

28.5.239 assoc-list/get-keys

[assoc-list] [Methods]

DESCRIPTION:

 Get a simple list of the keys in a given association list.

ARGUMENTS:

 - An assoc-list.

OPTIONAL ARGUMENTS:

 - Optional argument: T or NIL (default T) to indicate whether a warning
   should be printed when the first argument is a recursive assoc-list.

RETURN VALUE:

 A list of the keys only of all top-level association list pairs in the
 given assoc-list.

 get-keys is a method of the assoc-list class and therefore returns only
 top-level keys if accessing a recursive assoc-list.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog fido)
                                   (cow bessie)))))
  (get-keys al))

=> (CAT DOG COW)

(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog ((scottish terrier)
                                         (german shepherd)
                                         (irish wolfhound)))
                                   (cow bessie)))))
  (get-keys al))

=> (CAT DOG COW)
```

SYNOPSIS:

```
(defmethod get-keys ((al assoc-list) &optional (warn t))
```

28.5.240   assoc-list/get-nearest

[assoc-list] [Methods]

DATE:

 4/8/2015

DESCRIPTION:

 In assoc-lists with numeric IDs, return the object with the closest ID to
 the given key.

ARGUMENTS:

 - they key (number)
 - the assoc-list object

RETURN VALUE:

 a named-object

SYNOPSIS:

(defmethod get-nearest (key (al assoc-list) &optional ignore1 ignore2)

28.5.241   assoc-list/get-position

[assoc-list] [Methods]

DESCRIPTION:

 Returns the index position (zero-based) of a named-object within a given
 assoc-list.

ARGUMENTS:

 - The assoc-list key symbol (named-object id) of the object for which the
   position is sought.
 - The assoc-list in which it is to be sought.

OPTIONAL ARGUMENTS:

 - Optional argument: An indexing integer. In this case, get-position will
   search for the given object starting part-way into the list, skipping all
   objects located at indices lower than the given integer (default = 0).

RETURN VALUE:

 The integer index of the named-object within the given assoc-list.

 NIL is returned if the object is not present in the assoc-list starting
 with the index number given as the start argument (i.e., in the entire list
 if the optional start argument is omitted).

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-position 'four al))

=> 1

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-position 'jack al))

=> NIL

(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-position 'jim al 1))

=> NIL
```

SYNOPSIS:

```
(defmethod get-position (key (al assoc-list) &optional (start 0))
```

28.5.242   assoc-list/l-for-lookup

[assoc-list] [Classes]

NAME:

 l-for-lookup

 File: l-for-lookup

Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> l-for-lookup

Version: 1.0.11

Project: slippery chicken (algorithmic composition)

Purpose: Implementation of the l-for-lookup class. The name
                  stands for L-System for Lookups (L for
                  Lindenmayer). This provides an L-System
                  function for generating sequences of numbers
                  from rules and seeds, and then using these
                  numbers for lookups into the assoc-list. In the
                  assoc list are stored groups of numbers, meant
                  to represent in the first place, for example,
                  rhythmic sequences. The grouping could be as
                  follows: ((2 3 7) (11 12 16) (24 27 29) and
                  would mean that a transition should take place
                  (over the length of the number of calls
                  represented by the number of L-Sequence results)
                  from the first group to the second, then from
                  the second to the third. When the first group
                  is in use, then we will simple cycle around the
                  given values, similar with the other groups.
                  The transition is based on a fibonacci algorithm
                  (see below).

                  The sequences are stored in the data slot. The l-sequence
                  will be a list like (3 1 1 2 1 2 2 3 1 2 2 3 2 3 3 1).
                  These are the references into the assoc-list (the 1, 2, 3
                  ids in the list below).

                  e.g. ((1 ((2 3 7) (11 16 12)))
                        (2 ((4 5 9) (13 14 17)))
                        (3 ((1 6 8) (15 18 19))))

Author: Michael Edwards: m@michael-edwards.org

Creation date: 15th February 2002

$$ Last modified: 17:07:59 Thu May 28 2020 CEST

SVN ID: Id

28.5.243   l-for-lookup/count-elements

[l-for-lookup] [Functions]

DESCRIPTION:

 Count the number of times each element occurs in a given list.

ARGUMENTS:

 - A list of numbers or symbols (or anything which can be compared using
   EQL).

RETURN VALUE:

 Returns a list of two-element lists, each consisting of one list
 element from the specified list and the number of times that element occurs
 in the list. If the elements are numbers, these will be sorted from low to
 high, otherwise symbols will be returned from most populous to least.

EXAMPLE:

```
(count-elements '(1 4 5 7 3 4 1 5 4 8 5 7 3 2 3 6 3 4 5 4 1 4 8 5 7 3 2))
```

```
=> ((1 3) (2 2) (3 5) (4 6) (5 5) (6 1) (7 3) (8 2))
```

SYNOPSIS:

```
(defun count-elements (list)
```

28.5.244   l-for-lookup/do-lookup

[l-for-lookup] [Methods]

DESCRIPTION:

 Generate an L-sequence from the rules of the specified l-for-lookup object
 and use it to perform the Fibonacci-based transitioning look-up of values in
 the specified sequences.

 NB As this accesses the rules as circular lists whose states are not reset
 before starting, you may get (desired) different results if you call the
 routine more than once. In order to avoid that call (reset lflu) before
 do-lookup.

ARGUMENTS:

  - An l-for-lookup object.
  - The start seed, or axiom, that is the initial state of the L-system. This
    must be the key-id of one of the sequences.
  - An integer that is the length of the sequence to be returned. NB: This
    number does not indicate the number of L-system passes, but only the
    number of elements in the list returned, which may be the first segment
    of a sequence returned by a pass that actually generates a much longer
    sequence.

OPTIONAL ARGUMENTS:

  - A number which is the factor by which returned numerical values are to be
    scaled. If NIL, the method will use the value in the given l-for-lookup
    object's SCALER slot instead. Default = NIL. NB: The value of the given
    l-for-lookup object's OFFSET slot is additionally used to increase
    numerical values before they are returned.

RETURN VALUE:

 This method returns three lists:
 - The resulting sequence.
 - The distribution of the values returned by the look-up.
 - The L-sequence of the key-IDs.

EXAMPLE:

```
;; Create an l-for-lookup object in which the sequences are defined such that
;; the transition takes place over the 3 given lists and from x to y to z, and
;; apply the do-lookup method to see the results. Each time one of these lists
;; is accessed, it will cyclically return the next value.
(let ((lfl (make-l-for-lookup
            'lfl-test
            '((1 ((ax1 ax2 ax3) (ay1 ay2 ay3 ay4) (az1 az2 az3 az4 az5)))
              (2 ((bx1 bx2 bx3) (by1 by2 by3 by4) (bz1 bz2 bz3 bz4 bz5)))
              (3 ((cx1 cx2 cx3) (cy2 cy2 cy3 cy4) (cz1 cz2 cz3 cz4 cz5))))
            '((1 (1 2 2 2 1 1))
              (2 (2 1 2 3 2))
              (3 (2 3 2 2 2 3 3))))))
  (do-lookup lfl 1 211))

=>
(AX1 BX1 BX2 BX3 AX2 AX3 BX1 AX1 BX2 CX1 BX3 BX1 AX2 BX2 CX2 BX3 BX1 AX3 BX2
     CX3 BX3 AX1 BY1 BX1 BX2 AY1 AX2 AX3 BX3 BX1 BX2 AX1 AX2 BX3 AY2 BX1 CX1
     BY2 AX3 BX2 BX3 BX1 AX1 AY3 BX2 AX2 BY3 CY2 BX3 BX1 CX2 BY4 BX2 BX3 CX3
     CY2 BY1 AY4 BX1 CX1 BY2 BX2 AY1 BY3 CY3 BY4 AX3 BX3 BY1 BX1 AY2 AX1 BY2
```

```
      AY3 BY3 CY4 BX2 BY4 CX2 BY1 BX3 BY2 CY2 CX3 BY3 AY4 BY4 CY2 BY1 BX1 AX2
      BY2 CY3 BY3 AY1 BY4 BY1 BY2 AY2 AY3 BY3 AY4 BY4 CY4 BY1 BY2 CY2 BY3 BY4
      BY1 CY2 CY3 BY2 AY1 BY3 CY4 BY4 AY2 BY1 BY2 BY3 AY3 AY4 BY4 AY1 BY1 CZ1
      BZ1 BY2 AY2 BY3 CY2 BY4 BY1 AY3 BY2 CY2 BY3 AY4 BY4 BZ2 BY1 AZ1 AY1 AY2
      BY2 BY3 BY4 AY3 AY4 AZ2 BZ3 BY1 BY2 AY1 AY2 BZ4 AZ3 BY3 CZ2 BY4 BZ5 AY3
      BY1 CY3 BZ1 BY2 AZ4 BZ2 CZ3 BZ3 AZ5 BY3 BZ4 BY4 AY4 AZ1 AY1 BZ5 BZ1 BY1
      AZ2 AZ3 BZ2 AY2 BZ3 CY4 BY2 AZ4 BZ4 BZ5 BZ1 AZ5 AZ1 BZ2 AZ2 BY3 CZ4 BZ3
      BZ4 CY2 BZ5 BZ1 BZ2 CZ5 CZ1 BZ3 AZ3 BZ4 CZ2 BZ5),
((CX1 3) (AX3 5) (AX1 6) (BX2 11) (CX2 3) (BX3 11) (CX3 3) (BX1 12) (AX2 6)
 (AY3 7) (CY3 4) (CZ3 1) (BY4 14) (AY4 7) (AY1 8) (BY1 15) (AY2 8) (CY4 4)
 (BY2 15) (AZ4 2) (AZ5 2) (AZ1 3) (AZ2 3) (BY3 15) (CZ4 1) (CY2 9) (BZ1 5)
 (BZ25) (CZ5 1) (CZ1 2) (BZ3 5) (AZ3 3) (BZ4 5) (CZ2 2) (BZ5 5)),
(1 2 2 2 1 1 2 1 2 3 2 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 1 2 2 2 1 1 2 1 2 3 2 1
   2 2 2 1 1 2 1 2 3 2 2 3 2 2 2 3 3 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 2 1 2 3 2
   2 3 2 2 2 3 3 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 2 1 2 3 2 2 3 2 2 2 3 3 2 1 2
   3 2 1 2 2 2 1 1 2 1 2 3 2 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 1 2 2 2 1 1 1 2 2
   2 1 1 2 1 2 3 2 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 1 2 2 2 1 1 2 1 2 3 2 1 2 2
   2 1 1 2 1 2 3 2 2 3 2 2 2 3 3 2 1 2 3 2)
```

SYNOPSIS:

```
(defmethod do-lookup ((lflu l-for-lookup) seed stop &optional scaler)
```

28.5.245   l-for-lookup/do-lookup-linear

[l-for-lookup] [Methods]

DESCRIPTION:

```
 Similar to do-lookup but here we generate a linear sequence (with
 get-linear-sequence) instead of an L-System.
```

ARGUMENTS:

```
 - An l-for-lookup object.
 - The start seed, or axiom, that is the initial state of the linear
   system. This must be the key-id of one of the sequences.
 - An integer that is the length of the sequence to be returned.
```

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :scaler. A number which is the factor by which returned numerical values
   are to be scaled. If NIL, the method will use the value in the given
   l-for-lookup object's SCALER slot instead. Default = NIL. NB: The value
```

  of the given l-for-lookup object's OFFSET slot is additionally used to
  increase numerical values before they are returned. Default = NIL.
 - :reset. T or NIL to indicate whether to reset the pointers of the given
   circular lists before proceeding. T = reset. Default = T.

RETURN VALUE:

 This method returns three lists:
 - The resulting sequence.
 - The distribution of the values returned by the look-up.
 - The L-sequence of the key-IDs.

EXAMPLE:

```
;; This will return the result of lookup, the number of repetitions of each
;; rule key in the result of lookup, and the linear-sequence itself.
(let* ((tune (make-l-for-lookup
              'tune
              '((1 ((2 1 8)))
                (2 ((3 4)))
                (3 ((4 5)))
                (4 ((5 1 6)))
                (5 ((6 5 7 4)))
                (6 ((4 5)))
                (7 ((4 5 1)))
                (8 ((1))))
              '((1 (1 2)) (2 (1 3 2)) (3 (1 4 3)) (4 (1 2 1)) (5 (5 3 1))
                (6 (2 5 6)) (7 (5 6 4)) (8 (3 2)))))))
  (do-lookup-linear tune 1 100))
=>
(1 3 4 1 4 4 6 8 2 1 1 4 5 5 6 5 1 1 7 6 8 2 1 1 3 5 4 4 5 5 6 5 4 7 1 4 4 6 8
 2 1 1 4 5 5 6 5 5 8 2 1 1 3 4 1 7 6 8 2 1 1 4 5 4 1 4 5 6 5 1 6 8 2 1 1 3 5 7
 5 4 1 4 5 6 5 4 7 6 8 2 1 1 4 5 4 1 4 5 6 5)
((1 24) (2 7) (3 4) (4 20) (5 21) (6 12) (7 5) (8 7))
(1 2 3 4 5 6 4 1 1 8 1 2 4 6 5 5 7 4 5 4 1 1 8 1 2 3 5 6 4 6 5 5 7 5 4 5 6 4 1
 1 8 1 2 4 6 5 5 7 1 1 8 1 2 3 4 5 4 1 1 8 1 2 4 6 4 5 6 5 5 7 4 1 1 8 1 2 3 5
 4 6 4 5 6 5 5 7 5 4 1 1 8 1 2 4 6 4 5 6 5 5)
```

SYNOPSIS:

```
(defmethod do-lookup-linear ((lflu l-for-lookup) seed stop
                             &key scaler (reset t))
```

28.5.246   l-for-lookup/do-simple-lookup

[l-for-lookup] [Methods]

DESCRIPTION:

Performs a simple look-up procedure whereby a given reference key always
returns a specific and single piece of data. This is different from
do-lookup, which performs a transitioning between lists and returns items
from those lists in a circular manner. do-simple-lookup always returns the
first element of the sequence list associated with a given key-ID.

N.B. the SCALER and OFFSET slots are ignored by this method.

ARGUMENTS:

- An l-for-lookup object.
- The start seed, or axiom, that is the initial state of the L-system. This
  must be the key-id of one of the sequences.
- An integer that is the number of elements to be returned.

RETURN VALUE: EXAMPLE:

```
;; Create an l-for-lookup object using three production rules and three
;; sequences of three lists. Applying do-simple-lookup returns the first
;; element of each sequence based on the L-sequence of keys created by the
;; rules of the give l-for-lookup object.
(let ((lfl (make-l-for-lookup
             'lfl-test
             '((1 ((ax1 ax2 ax3) (ay1 ay2 ay3 ay4) (az1 az2 az3 az4 az5)))
               (2 ((bx1 bx2 bx3) (by1 by2 by3 by4) (bz1 bz2 bz3 bz4 bz5)))
               (3 ((cx1 cx2 cx3) (cy2 cy2 cy3 cy4) (cz1 cz2 cz3 cz4 cz5))))
             '((1 (1 2 2 2 1 1))
               (2 (2 1 2 3 2))
               (3 (2 3 2 2 2 3 3))))))
  (do-simple-lookup lfl 1 21))

=> ((AX1 AX2 AX3) (BX1 BX2 BX3) (BX1 BX2 BX3) (BX1 BX2 BX3) (AX1 AX2 AX3)
    (AX1 AX2 AX3) (BX1 BX2 BX3) (AX1 AX2 AX3) (BX1 BX2 BX3) (CX1 CX2 CX3)
    (BX1 BX2 BX3) (BX1 BX2 BX3) (AX1 AX2 AX3) (BX1 BX2 BX3) (CX1 CX2 CX3)
    (BX1 BX2 BX3) (BX1 BX2 BX3) (AX1 AX2 AX3) (BX1 BX2 BX3) (CX1 CX2 CX3)
    (BX1 BX2 BX3))
```

SYNOPSIS:

```
(defmethod do-simple-lookup ((lflu l-for-lookup) seed stop)
```

28.5.247   l-for-lookup/fibonacci

[l-for-lookup] [Functions]

DESCRIPTION:

Return the longest possible list of sequential Fibonacci numbers whose
combined sum is less than or equal to the specified value. The list is
returned in descending sequential order, ending with 0.

The function also returns as a second individual value the first Fibonacci
number that is greater than the sum of the list returned.

NB: The value of the second number returned will always be equal to the sum
    of the list plus one. In most cases that number will be less than the
    number specified as the argument to the fibonacci function, and
    sometimes it will be equal to the number specified; but in cases where
    the sum of the list returned is equal to the number specified, the
    second number returned will be equal to the specified number plus one.

ARGUMENTS:

A number that is to be the test number.

RETURN VALUE:

A list of descending sequential Fibonacci numbers, of which list the last
element is 0.

Also returns as a second individual value the first Fibonacci number that
is greater than the sum of the list returned, which will always be the sum
of that list plus one.

EXAMPLE:

```
;; Returns a list of consecutive Fibonacci numbers from 0 whose sum is equal to
;; or less than the value specified. The second number returned is the first
;; Fibonacci number whose value is greater than the sum of the list, and will
;; always be the sum of the list plus one.
(fibonacci 5000)

 => (1597 987 610 377 233 144 89 55 34 21 13 8 5 3 2 1 1 0), 4181

;; The sum of the list
(+ 1597 987 610 377 233 144 89 55 34 21 13 8 5 3 2 1 1 0)

=> 4180
```

SYNOPSIS:

```
(defun fibonacci (max-sum)
```

28.5.248   l-for-lookup/fibonacci-divide

[l-for-lookup] [Functions]

DATE:

 August 5th 2019, Heidhausen

DESCRIPTION:

 Divide an integer into ever smaller proportions as determined by a Fibonacci
 series which sums to the largest Fibonacci number less than the given
 integer. The resulting list is of ascending (accumulating) numbers which are
 scaled to fit the given integer (optionally but by default) and (optionally
 but also by default) rounded. As such, this routine is useful for
 determining, for examples, bar numbers at which events/changes might
 happen.

ARGUMENTS:

 - an integer

OPTIONAL ARGUMENTS:

 - T or NIL: whether to scale the Fibonacci numbers to fit the given
 integer. Default = T.
 - T or NIL: whether to round after scaling

RETURN VALUE:

 A list of numbers

EXAMPLE:

```
(FIBONACCI-DIVIDE 300)
--> (116 187 230 258 274 285 291 295 297 299 300)

;;; don't scale to 300 (i.e. only increase by Fibonacci numbers but note that
;;; the Fibonacci number is added to the current number, which starts at one,
;;; hence 1+89, 90+55, 145+34 ...
(FIBONACCI-DIVIDE 300 nil)
--> (90 145 179 200 213 221 226 229 231 232 233)

;;; do scale but don't round
(FIBONACCI-DIVIDE 300 t nil)
--> (115.87983 186.69528 230.4721 257.51074 274.24893 284.54935 290.98712
     294.8498 297.4249 298.71246 300.0)
```

SYNOPSIS:

```
(defun fibonacci-divide (num &optional (scale t) (round t))
```

28.5.249 l-for-lookup/fibonacci-start-at-2

[l-for-lookup] [Functions]

DESCRIPTION:

Return the longest possible list of sequential Fibonacci numbers, excluding
0 and 1, whose combined sum is less than or equal to the specified
value. The list is returned in descending sequential order.

The function also returns as a second value the sum of the list.

NB: In addition to excluding 0 and 1, this function also differs from the
    plain fibonacci function in that the second value returned is the sum
    of the list rather than the first Fibonacci number greater than that
    sum.

ARGUMENTS:

A number that is to be the test number.

RETURN VALUE:

A list of descending sequential Fibonacci numbers, of which list the last
element is 2.

Also returns as a second result the sum of the list.

EXAMPLE:

```
;; Returns a list whose sum is less than or equal to the number specified as
;; the function's only argument
(fibonacci-start-at-2 17)

=> (5 3 2), 10

(fibonacci-start-at-2 20)

=> (8 5 3 2), 18

;; Two examples showing the different results of fibonacci
```

```
;; vs. fibonacci-start-at-2

;; 1
(fibonacci 18)

=> (5 3 2 1 1 0), 13

(fibonacci-start-at-2 18)

=> (8 5 3 2), 18

;; 2
(fibonacci 20)

=> (8 5 3 2 1 1 0), 21

(fibonacci-start-at-2 20)

=> (8 5 3 2), 18
```

SYNOPSIS:

```
(defun fibonacci-start-at-2 (max-sum)
```

28.5.250   l-for-lookup/fibonacci-transition

[l-for-lookup] [Functions]

DESCRIPTION:

```
Uses Fibonacci relationships to produces a sequence that is a gradual
transition from one repeating state to a second over n repetitions. The
function gradually increases the frequency of the second repeating state
until it completely dominates.

NB: The similar but separate function fibonacci-transition-aux1 gradually
    decreases state 1 and increases state 2.
```

ARGUMENTS:

```
 - An integer that is the desired number of elements in the resulting list
   (i.e., the number of repetitions over which the transition is to occur).
```

OPTIONAL ARGUMENTS:

- Repeating item 1 (starting state). This can be any Lisp type, including
  lists. Default = 0.
- Repeating item 2 (target state): This can also be any Lisp type.
  Default = 1.
- T or NIL to make a morph from item1 to item2. This means morphing items
  will be a morph structure. See below for an example. Default = NIL.
- T or NIL to make the morph first go down from item2 to item1 (mainly used
  by fibonacci-transitions). This can also be an envelope (see
  fibonacci-transitions for details). Default = NIL.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; Defaults to 0 and 1 (no optional arguments)
(fibonacci-transition 31)
=> (0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1)
;; Using optional arguments set to numbers
(fibonacci-transition 23 11 37)
=> (11 11 11 11 37 11 11 37 11 37 11 37 11 37 37 11 37 11 37 11 37 37 37)
;; Using lists
(fibonacci-transition 27 '(1 2 3) '(5 6 7))
=> ((1 2 3) (1 2 3) (1 2 3) (1 2 3) (5 6 7) (1 2 3) (1 2 3) (5 6 7) (1 2 3)
    (1 2 3) (5 6 7) (1 2 3) (5 6 7) (1 2 3) (5 6 7) (1 2 3) (5 6 7) (5 6 7)
    (1 2 3) (5 6 7) (5 6 7) (1 2 3) (5 6 7) (5 6 7) (5 6 7) (5 6 7) (5 6 7))
;; with morph: first going up then up/down
(fibonacci-transition 31 0 1 t) ->
(0 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.14285714285714285d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.2857142857142857d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.42857142857142855d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.5714285714285714d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.7142857142857142d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.857142857142857d0) 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.8) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.6)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.4) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.7) 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.7) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.85) 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1 1 1 1)
;; morph goes down then up
(fibonacci-transition 31 0 1 t t) -->
(0 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.6666666666666667d0)
 #S(MORPH :I1 0 :I2 1 :PROPORTION 0.3333333333333334d0)
```

```
#S(MORPH :I1 0 :I2 1 :PROPORTION 1.1102230246251565d-16)
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.25d0)
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.5d0)
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.75d0) 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.8) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.6)
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.4) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.7) 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.7) #S(MORPH :I1 0 :I2 1 :PROPORTION 0.85) 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1
#S(MORPH :I1 0 :I2 1 :PROPORTION 0.75) 1 1 1 1 1)
```

SYNOPSIS:

```
(defun fibonacci-transition (num-items &optional (item1 0) (item2 1) morph
                                       first-down)
```

28.5.251   l-for-lookup/fibonacci-transitions

[l-for-lookup] [Functions]

DATE:

 18 Feb 2010

DESCRIPTION:

 This function builds on the concept of the function fibonacci-transition by
 allowing multiple consecutive transitions over a specified number of
 repetitions. The function either produces sequences consisting of
 transitions from each consecutive increasing number to its upper
 neighbor, starting from 0 and continuing through a specified number of
 integers, or it can be used to produce a sequence by transitioning between
 each element of a user-specified list of items.

ARGUMENTS:

 - An integer indicating the number of repetitions over which the
   transitions are to be performed.
 - Either:
   - An integer indicating the number of consecutive values, starting from
     0, the function is to transition (i.e. 3 will produce a sequence that
     transitions from 0 to 1, then from 1 to 2 and finally from 2 to 3), or
   - A list of items of any type (including lists) through which the
     function is to transition.

OPTIONAL ARGUMENTS:

- T or NIL to make a morph from item1 to item2, 2 to 3, etc.. This means
  morphing items will be a morph structure. This can also be an envelope
  which thus allows morphing to be switched on and off over the course of
  the transition. The envelope can have any x range but the y-range should
  be from 0 to 1. Any value < 1 means the morph object will be ignored and
  either the :i1 or :i2 slot will be used instead, depending on whether the
  :proportion slot is > 0.5

RETURN VALUE:

A list.

EXAMPLE:

```
;; Using just an integer transitions from 0 to 1 below that integer
(fibonacci-transitions 76 4)

=> (0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 2 1 1 2 1 2 1
    2 2 1 2 1 2 2 2 2 2 2 2 3 2 2 3 2 3 2 3 3 2 3 2 3 3 3 2 3 3 3 3 3 3 3 3 3)

;; Using a list transitions consecutively through that list
(fibonacci-transitions 152 '(1 2 3 4))

=> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 2 1 2 1 2
    2 1 2 1 2 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2 3 2 2 3 2 3 2 3
    3 2 3 2 3 3 2 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 4 3 3 4 3 3 4 3 4 3 4
    4 3 4 3 4 4 3 4 4 3 4 4 4 4 4 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4)

;; A list of lists is also viable
(fibonacci-transitions 45 '((1 2 3) (4 5 4) (3 2 1)))

=> ((1 2 3) (1 2 3) (1 2 3) (1 2 3) (1 2 3) (4 5 4) (1 2 3) (1 2 3) (4 5 4)
    (1 2 3) (4 5 4) (1 2 3) (4 5 4) (1 2 3) (4 5 4) (1 2 3) (4 5 4) (1 2 3)
    (4 5 4) (4 5 4) (4 5 4) (4 5 4) (4 5 4) (3 2 1) (4 5 4) (3 2 1) (4 5 4)
    (3 2 1) (4 5 4) (3 2 1) (4 5 4) (3 2 1) (4 5 4) (3 2 1) (3 2 1) (3 2 1)
    (4 5 4) (3 2 1) (3 2 1) (3 2 1) (3 2 1) (3 2 1) (3 2 1) (3 2 1) (3 2 1))

;; using morph
(fibonacci-transitions 52 '(1 2 3) t)
=> (1 #S(MORPH :I1 1 :I2 2 :PROPORTION 0.16666666666666666d0)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.3333333333333333d0)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.5d0)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.6666666666666666d0)
```

```
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.8333333333333333d0) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.9)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.5d0)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.0d0)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.5d0) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.7)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.85) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2 2 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2 2 2 2
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.25d0)
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.5d0)
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75d0) 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.7)
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.85) 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3 3 3 3 3 3 3 3)

;; using morph and a morph envelope
(fibonacci-transitions 52 '(1 2 3 3)  '(0 0 10 1 52 1))
=> (1 1 1 1 2 2 2 2 1 1 1 2 #S(MORPH :I1 1 :I2 2 :PROPORTION 0.7)
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.85) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2 2 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2
      #S(MORPH :I1 1 :I2 2 :PROPORTION 0.75) 2 2 2 2 2 2 3 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.7)
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.85) 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3 3
      #S(MORPH :I1 2 :I2 3 :PROPORTION 0.75) 3 3 3 3 3 3 3 3 3)
```

SYNOPSIS:

```
(defun fibonacci-transitions (total-items levels &optional morph)
```

28.5.252   l-for-lookup/get-l-sequence

[l-for-lookup] [Methods]

DESCRIPTION:

 Return an L-sequence of the key-ids for the rules of a given l-for-lookup

object, created using the rules of that object. This method can be called
with an l-for-lookup object that contains no sequences, as it only returns
a list of the key-ids for the object's rules.

Tip: It seems that systems where one rule key yields all other keys as a
result makes for evenly distributed results which are different for each
seed.

ARGUMENTS:

- An l-for-lookup object.
- The start seed, or axiom, that is the initial state of the L-system. This
  must be the key-id of one of the sequences.
- An integer that is the length of the sequence to be returned. NB: This
  number does not indicate the number of L-system passes, but only the
  number of elements in the list returned, which may be the first segment
  of a sequence returned by a pass that actually generates a much longer
  sequence.

RETURN VALUE:

A list that is the L-sequence of rule key-ids.

The second value returned is a count of each of the rule keys in the
sequence created, in their given order.

EXAMPLE:

```
;; Create an l-for-lookup object with three rules and generate a new sequence
;; of 29 rule keys from those rules. The l-for-lookup object here has been
;; created with the SEQUENCES argument set to NIL, as the get-l-sequence
;; method requires no sequences. The second list returned indicates the
;; number of times each key appears in the resulting sequence (thus 1 appears 5
;; times, 2 appears 12 times etc.)
(let ((lfl (make-l-for-lookup 'lfl-test
                              NIL
                              '((1 (2))
                                (2 (1 3))
                                (3 (3 2))))))
  (get-l-sequence lfl 1 29))

=> (2 3 2 3 2 1 3 2 3 2 3 2 1 3 2 3 2 1 3 3 2 1 3 2 3 2 3 2 1), (5 12 12)

;; A similar example using symbols rather than numbers as keys and data
(let ((lfl (make-l-for-lookup 'lfl-test
```

```
                                        NIL
                                        '((a (b))
                                          (b (a c))
                                          (c (c b))))))
  (get-l-sequence lfl 'a 19))
```

=> (A C C B A C C B A C B C B A C C B A C), (5 5 9)

SYNOPSIS:

```
(defmethod get-l-sequence ((lflu l-for-lookup) seed stop-length)
```

28.5.253   l-for-lookup/get-linear-sequence

[l-for-lookup] [Methods]

DESCRIPTION:

 Instead of creating L-sequences with specified rules, use them
 generate a simple sequential list.

 The method first returns the first element in the list whose ID matches the
 SEED argument, then that element is used as the ID for the next
 look-up. Each time a rule is accessed, the next element in the rule
 is returned (if there is more than one), cycling to the head of the list
 once its end is reached.

 Seen very loosely, this method functions a bit like a first-order Markov
 chain, but without the randomness.

ARGUMENTS:

 - An l-for-lookup object.
 - The seed, which is the starting key for the resulting sequence. This must
   be the key-ID of one of the sequences.
 - An integer that is the number of elements to be in the resulting list.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to reset the pointers of the given circular
   lists before proceeding. T = reset. Default = T.

RETURN VALUE:

 A list of results of user-defined length.

EXAMPLE:

```
(let ((lfl (make-l-for-lookup 'lfl-test
                              nil
                              '((1 (2 3))
                                (2 (3 1 2))
                                (3 (1))))))
  (get-linear-sequence lfl 1 23))

=> (1 2 3 1 3 1 2 1 3 1 2 2 3 1 3 1 2 1 3 1 2 2 3)
```

SYNOPSIS:

```
(defmethod get-linear-sequence ((lflu l-for-lookup) seed stop-length
                                &optional (reset t))
```

28.5.254   l-for-lookup/make-l-for-lookup

[l-for-lookup] [Functions]

DESCRIPTION:

Create an l-for-lookup object. The l-for-lookup object uses techniques
associated with Lindenmayer-systems (or L-systems) by storing a series of
rules about how to produce new, self-referential sequences from the data of
original, shorter sequences.

NB: This method just stores the data concerning sequences and rules. To
    manipulate the data and create new sequences, see do-lookup or
    get-l-sequence etc.

ARGUMENTS:

- A symbol that will be the object's ID.
- A sequence (list) or list of sequences, that serve(s) as the initial
  material, from which the new sequence is to be produced.
- A production rule or list of production rules, each consisting of a
  predecessor and a successor, defining how to expand and replace the
  individual predecessor items.

OPTIONAL ARGUMENTS:

keyword arguments:
- :auto-check-redundancy. Some rules (e.g. in simplest form, a set of keys
  that return only one result each) we'll get a result of the same length no

matter how many results we ask for. This can sometimes be problematic so
do a check if requested. Default = NIL.
- :scaler. Factor by which to scale the values returned by
  do-lookup. Default = 1. Does not modify the original data.
- :offset. Number to be added to values returned by do-lookup (after they
  are scaled). Default = NIL. Does not modify the original data.

RETURN VALUE:

Returns an l-for-lookup object.

EXAMPLE:

```
;; Create an l-for-lookup object based on the Lindenmayer rules (A->AB) and
;; (B->A), using the defaults for the keyword arguments
(make-l-for-lookup 'l-sys-a
                   '((1 ((a)))
                     (2 ((b))))
                   '((1 (1 2)) (2 (1))))

=>
L-FOR-LOOKUP:
[...]
              l-sequence: NIL
              l-distribution: NIL
              ll-distribution: NIL
              group-indices: NIL
              scaler: 1
              offset: 0
              auto-check-redundancy: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: L-SYS-A, tag: NIL,
data: (
[...]

;; A larger list of sequences, with keyword arguments specified
(make-l-for-lookup 'lfl-test
                                '((1 ((2 3 4) (5 6 7)))
                                  (2 ((3 4 5) (6 7 8)))
                                  (3 ((4 5 6) (7 8 9))))
                                '((1 (3)) (2 (3 1)) (3 (1 2)))
                                :scaler 1
                                :offset 0
```

```
                                      :auto-check-redundancy nil)
```

SYNOPSIS:

```
(defun make-l-for-lookup (id sequences rules &key (auto-check-redundancy nil)
                                             (offset 0)
                                             (scaler 1))
```

28.5.255   l-for-lookup/remix-in

[l-for-lookup] [Functions]

DESCRIPTION:

 Given a list (for example generated by fibonacci-transitions) where we
 proceed sequentially through adjacent elements, begin occasionally mixing
 earlier elements of the list back into the original list once we've reached
 the third unique element in the original list.

 The earlier elements are mixed back in sequentially (the list is mixed back
 into itself), starting at the beginning of the original list, and inserted
 at automatically selected positions within the original list.

 This process results in a longer list than the original, as earlier
 elements are spliced in, without removing the original elements and their
 order. If however the :replace keyword is set to T, then at the selected
 positions those original elements will be replaced by the earlier
 elements. This could of course disturb the appearance of particular
 results and patterns.

 The :remix-in-fib-seed argument determines how often an earlier element is
 re-inserted into the original list. The lower the number, the more often an
 earlier element is mixed back in. A value of 1 or 2 will result in each
 earlier element being inserted after every element of the original (once
 the third element of the original has been reached).

 NB: The affects of this method are less evident on short lists.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

 keyword arguments:

- :remix-in-fib-seed. A number that indicates how frequently an earlier
  element will be mixed back into the original list. The higher the number,
  the less often earlier elements are remixed in. Default = 13.
- :mirror. T or NIL to indicate whether the method should pass backwards
  through the original list once it has reached the end. T = pass
  backwards. Default = NIL.
- :test. The function used to determine the third element in the list. This
  function must be able to compare whatever data type is in the
  list. Default = #'eql.
- :replace. If T, retain the original length of the list by replacing items
  rather than splicing them in (see above). Default = NIL.

RETURN VALUE:

 Returns a new list.

EXAMPLE:

```
;; Straightforward usage with default values
(remix-in (fibonacci-transitions 320 '(1 2 3 4 5)))

=> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2
    1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2 1 2 2 1
    2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 1 2 2 1 2 2 1 2 1 3
    1 2 2 1 2 1 2 1 3 2 2 1 3 2 1 2 3 1 2 1 3 1 2 3 1 2 1 3 1 2 3 2 1 3 2 1 3 2
    1 3 1 3 1 2 3 1 3 1 2 1 3 3 3 2 3 2 1 3 3 1 3 1 3 1 3 3 1 3 1 3 1 3 3 3 3 1 3
    3 1 3 3 1 3 1 3 1 3 3 2 3 1 4 1 3 3 3 1 3 3 1 3 3 1 4 1 3 1 3 3 2 3 1 4 1 3
    3 4 1 3 3 1 4 3 2 4 1 3 1 4 3 2 4 1 3 1 4 3 4 2 3 4 1 3 4 2 4 1 3 2 4 4 1 3
    2 4 1 4 4 4 2 3 4 1 4 4 2 4 1 4 2 4 4 1 4 2 4 2 4 4 4 1 4 4 2 4 4 2 4 1 4 2
    4 4 2 5 2 4 2 4 4 4 1 4 4 2 4 5 2 4 2 4 2 4 4 2 5 2 4 2 4 5 4 2 4 5 2 4 5 2
    4 2 5 2 4 5 2 4 2 5 2 4 5 4 2 5 4 2 5 5 2 4 2 5 2 5 4 3 5 2 5 2 5 5 4 2 5 5
    2 5 5 2 5 2 5 2 5 5 3 4 2 5 2 5 5 5 2 5 5 2 5 5 3 5 2 5 2 5 5 3 5 2 5 2 5 5
    5 3 5 5 2 5 5 3 5 2 5 3 5 5 2 5 3 5 2 5 5 5 3 5 5 2 5 5 3 5 2 5 3 5 5 2)

;; A lower :remix-in-fib-seed value causes the list to be mixed back into
;; itself at more frequent intervals
(remix-in (fibonacci-transitions 320 '(1 2 3 4 5)) :remix-in-fib-seed 3)

=> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2
    1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2 1 2 2 1
    2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 1 2 1 2 2 1 2 1 2 2 1
    3 1 2 2 1 2 1 2 1 2 3 1 2 1 2 3 1 2 1 2 3 1 2 1 3 2 1 3 1 2 3 1 2 1 3 2 1 3 1 2
    3 1 2 1 3 3 1 2 1 3 3 2 2 1 3 3 1 3 1 3 2 1 3 1 3 3 1 3 1 3 3 1 3 1 3 3 3 1 3
    1 3 3 1 3 2 3 3 1 3 1 3 3 1 3 1 3 4 1 3 1 3 3 1 3 2 3 3 1 3 1 4 3 1 3 1 3 3
    2 4 1 3 3 1 4 2 3 3 1 4 1 3 4 2 3 1 4 3 2 4 1 3 4 2 3 1 4 3 2 4 1 3 4 2 4 1
    3 4 2 4 1 3 4 2 4 1 4 4 2 3 2 4 4 1 4 2 4 4 2 4 1 4 4 2 4 2 4 4 4 2 4 2 4 4 1
```

```
    4 2 4 4 2 4 2 4 4 2 5 2 4 4 2 4 2 4 4 2 4 2 4 5 2 4 2 4 4 2 4 2 5 4 2 4 2 5
    4 2 4 2 5 4 2 5 2 4 5 2 4 3 5 4 2 5 2 4 5 2 4 2 5 4 2 5 2 5 4 2 5 3 5 4 2 5
    2 5 5 2 5 2 4 5 3 5 2 5 5 2 5 3 5 5 2 5 2 4 5 3 5 2 5 5 3 5 2 5 5 3 5 2 5 5
    3 5 2 5 5 3 5 2 5 5 3 5 2 5 5 3 5 2 5 5 3 5 3 5 5 2 5 3 5 5 3 5 2 5 5 3 5 3
    5 5 3 5 3 5 5 2 5 3)
```

;; Setting the keyword argument <mirror> to T causes the method to reverse back
;; through the original list after the end has been reached
(remix-in (fibonacci-transitions 320 '(1 2 3 4 5))
          :remix-in-fib-seed 3
          :mirror t)

```
=> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 2
    1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2 1 2 2 1
    2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 1 2 1 2 2 1 2 1 2 2 1
    3 1 2 2 1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 3 2 1 3 1 2 3 1 2 1 3 2 1 3 1 2
    3 1 2 1 3 3 1 2 1 3 3 2 2 1 3 3 1 3 1 3 2 1 3 1 3 3 1 3 1 3 3 1 3 1 3 3 1 3
    1 3 3 1 3 2 3 3 1 3 1 3 3 1 3 1 3 4 1 3 1 3 3 1 3 2 3 3 1 3 1 4 3 1 3 1 3 3
    2 4 1 3 3 1 4 2 3 3 1 4 1 3 4 2 3 1 4 3 2 4 1 3 4 2 3 1 4 3 2 4 1 3 4 2 4 1
    3 4 2 4 1 3 4 2 4 1 4 4 2 3 2 4 4 1 4 2 4 4 2 4 1 4 4 2 4 2 4 4 2 4 2 4 4 1
    4 2 4 4 2 4 2 4 4 2 5 2 4 4 2 4 2 4 4 2 4 2 4 5 2 4 2 4 4 2 4 2 5 4 2 4 2 5
    4 2 4 2 5 4 2 5 2 4 5 2 4 3 5 4 2 5 2 4 5 2 4 2 5 4 2 5 2 5 4 2 5 3 5 4 2 5
    2 5 5 2 5 2 4 5 3 5 2 5 5 2 5 3 5 5 2 5 2 4 5 3 5 2 5 5 3 5 2 5 5 3 5 2 5 5
    3 5 2 5 5 3 5 2 5 5 3 5 2 5 5 3 5 2 5 5 3 5 3 5 5 2 5 3 5 5 3 5 2 5 5 3 5 3
    5 5 3 5 3 5 5 2 5 3 5 5 3 5 3 5 5 3 5 3 5 5 3 5 3 5 5 3 5 3 5 5 3 5 3 5 5 3
    5 3 5 5 3 5 3 5 5 3 5 3 5 5 3 5 3 5 5 3 5 4 5 5 3 5 3 5 5 3 5 3 5 4 3 5 3 5
    5 3 5 4 5 5 3 5 3 5 4 3 5 3 5 5 4 5 3 4 5 3 5 4 4 5 3 5 3 4 5 4 4 3 5 4 4 5
    3 4 5 4 4 3 5 4 4 5 3 4 5 4 4 3 4 5 4 4 3 4 5 4 4 3 4 4 4 4 4 5 4 3 4 4 4 4
    4 4 3 4 4 4 5 4 4 4 4 4 4 4 4 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
    4 4 4 3 4 4 4 4 4 4 4 4 3 4 4 4 4 3 4 4 4 4 3 4 4 4 3 4 4 3 5 4 3 4 4 4 3 4 4
    3 4 4 3 4 4 4 3 3 4 4 5 3 3 4 4 4 3 3 4 3 4 3 4 5 3 4 3 3 4 3 5 3 3 4 3 4 4
    3 5 3 4 3 3 5 3 4 3 3 5 3 4 3 3 5 3 4 3 3 5 3 4 3 3 5 3 4 3 3 5 3 4 2 3 5 3
    5 3 3 4 2 5 3 3 5 2 4 3 3 5 2 5 3 2 5 3 5 2 3 4 2 5 3 2 5 3 5 2 3 5 2 5 3 2
    5 2 5 3 2 5 2 4 3 2 5 2 5 2 2 5 3 5 2 2 5 2 5 2 2 5 2 5 2 3 5 2 5 2 2 5 2 5
    2 2 5 2 5 2 2 5 2 5 2 2 5 2 5 2 2 5 2 5 2 2 5 2 5 2 1 5 2 5 2 2 5 2 5 1 2 5
    2 5 1 2 5 2 5 1 2 5 1 5 2 1 5 2 5 1 2 5 1 5 2 1 5 2 5 1 2 5 1 5 1 2 5 1 5 1
    2 5 1 5 1 1 5 1 5 2 1 5 1 5 1 1 5 1 5 1 1 5 2 5 1 1 5 1 5 1 1 5 1 5 1 1 5 1
    5 1 1 5 1 5 2 1 5 1 5 1 1 5 1 5 1 1 5 1 5 1 1 5 1 5 1 1 5 1 5 1 1 5 1 5 1 1
    5 1 5 1 1 5 1 4 1)
```

SYNOPSIS:

```
(defun remix-in (list &key (remix-in-fib-seed 13) (mirror nil) (test #'eql)
                      (replace nil))
```

28.5.256   l-for-lookup/reset

[l-for-lookup] [Methods]

DESCRIPTION:

Sets the counters (index pointers) of all circular-sclist objects stored
within a given l-for-lookup object back to zero.

ARGUMENTS:

- An l-for-lookup object.

OPTIONAL ARGUMENTS:

- (an optional IGNORE argument for internal use only).

RETURN VALUE:

Always T.

SYNOPSIS:

```
(defmethod reset ((lflu l-for-lookup) &optional ignore1 ignore2)
```

28.5.257   assoc-list/make-assoc-list

[assoc-list] [Functions]

DESCRIPTION:

A function that provides a shortcut to creating an assoc-list, filling it
with data, and assigning a name to it.

ARGUMENTS:

- The name of the assoc-list to be created.
- The data with which to fill it.

OPTIONAL ARGUMENTS:

keyword arguments:
- :warn-not-found. T or NIL to indicate whether a warning is printed when an
  index which doesn't exist is used for look-up. T = warn. Default = T.

RETURN VALUE:

 Returns the assoc-list as a named-object.

EXAMPLE:

```
(make-assoc-list 'looney-tunes '((bugs bunny)
                                 (daffy duck)
                                 (porky pig)))
=>
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL
                     this: NIL
                     next: NIL
NAMED-OBJECT: id: LOONEY-TUNES, tag: NIL,
data: (
NAMED-OBJECT: id: BUGS, tag: NIL,
data: BUNNY

NAMED-OBJECT: id: DAFFY, tag: NIL,
data: DUCK

NAMED-OBJECT: id: PORKY, tag: NIL,
data: PIG)
```

SYNOPSIS:

```
(defun make-assoc-list (id al &key (warn-not-found t))
```

28.5.258   assoc-list/map-data

[assoc-list] [Methods]

DESCRIPTION:

 Map a function over the data in the assoc-list, i.e. call the given
 function/method on each data slot of each named-object in the assoc-list
 data slot. See also recursive-assoc-list's rmap method which does pretty
 much the same but acting recursively on each named-object (unless it is
 itself recursive), rather than the named-object's data.

ARGUMENTS:

- The assoc-list to which the function is to be applied.
- The function to be applied. This must take the data in the assoc-list as
  a first argument.

OPTIONAL ARGUMENTS:

- &rest: Further arguments for the function.

RETURN VALUE:

 Returns a list of the values returned by the function call on the data.

EXAMPLE:

```
(let ((al (make-assoc-list 'al-test
                           '((1 (1 2 3 4))
                             (2 (5 6 7 8))
                             (3 (9 10 11 12))))))
  (map-data al #'(lambda (y)
                   (loop for i in y collect (* i 2)))))

=> ((2 4 6 8) (10 12 14 16) (18 20 22 24))
```

SYNOPSIS:

```
(defmethod map-data ((al assoc-list) function &rest further-arguments)
```

28.5.259   assoc-list/nmap-data

[assoc-list] [Methods]

DATE:

 November 1st 2018, Heidhausen

DESCRIPTION:

 A destructive version of map-data: replace the data slot with the
 named-object returned by the 2nd argument after it is passed one
 named-object after another.

ARGUMENTS:

 See the map-data method in this class.

RETURN VALUE:

 the assoc-list object with modified data in its elements.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((1 (2 3))
                                   (2 (4 5 6))
                                   (3 (7))))))
  (nmap-data al #'(lambda (no) (length (data no))))
  al)
...
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data: 2
**************
NAMED-OBJECT: id: 2, tag: NIL,
data: 3
**************
NAMED-OBJECT: id: 3, tag: NIL,
data: 1
...
```

SYNOPSIS:

```
(defmethod nmap-data ((al assoc-list) function &rest further-arguments)
```

28.5.260   assoc-list/recursive-assoc-list

[assoc-list] [Classes]

NAME:

 recursive-assoc-list

 File: recursive-assoc-list.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Extension of the assoc-list class to allow and

```
                    automatically instantiate association lists inside of
                    association lists to any level of nesting. E.g.
                    (setf x
                      '((1 one)
                        (2 two)
                        (3 ((cat "cat")
                            (dog ((mickey mouse)
                                  (donald duck)
                                  (daffy duck)
                                  (uncle ((james dean)
                                          (dean martin)
                                          (fred astaire)
                                          (ginger ((wolfgang mozart)
                                                   (johann bach)
                                                   (george gershwin)))))))
                            (mouse "mouse")))
                        (4 four)))
```

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: March 18th 2001

 $$ Last modified: 11:04:58 Sat Sep 19 2020 CEST

 SVN ID: Id

28.5.261   recursive-assoc-list/add

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Add a new element (key/data pair) to the given recursive-assoc-list
 object.

 If no value is specified for the optional argument, the new element is
 added at the end of the top level. The optional argument allows for the
 FULL-REF to be specified, i.e. a recursive path of keys down to the nested
 level where the new element is to be placed.

 N.B. All new items are added to the end of the data list, unlike in
 assoc-list where that is the default but can be changed. Here it cannot be
 changed.

ARGUMENTS:

 – A key/data pair.
 – A recursive-assoc-list object.

OPTIONAL ARGUMENTS:

 – A list that is the FULL-REF, i.e. a recursive path of keys, down to the
   nested level where the new element is to be placed.

RETURN VALUE:

 Returns T if the specified named-object is successfully added to the given
 recursive-assoc-list.

 Returns an error if an attempt is made to add NIL to the given
 recursive-assoc-list or if the given named-object is already present at the
 same level within the given recursive-assoc-list.

EXAMPLE:

```
;; Adding an element while specifying no optional argument results in the new
;; element being placed at the end of the top level by default (evident here by
;; the fact that the ref for (MAKERS) is a single-item list)
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (add '(makers mark) ral)
  (get-all-refs ral))

=> ((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
    (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE)
    (MAKERS))

;; A list that is a path of keys (FULL-REF) to the desired recursive level must
;; be given as the optional argument in order to place the specified element
;; deeper in the given recursive-assoc-list object
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
```

```
                                        (violets ((blue velvet)
                                                   (red ((dragon den)
                                                          (viper nest)
                                                          (fox hole)))
                                                   (white ribbon)))))))))
  (add '(yellow sky) ral '(four violets))
  (get-all-refs ral))

=> ((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
    (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE)
    (FOUR VIOLETS YELLOW))

;; Attempting to add an element that is already present at the given level of
;; the given recursive-assoc-list object results in an error
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                         (red ((dragon den)
                                                (viper nest)
                                                (fox hole)))
                                         (white ribbon)))))))))
  (add '(makers mark) ral)
  (add '(makers mark) ral))

=>
assoc-list::add: Can't add MAKERS to assoc-list with id MIXED-BAG
    because key already exists!
   [Condition of type SIMPLE-ERROR]

;; Attempting to add NIL also results in an error
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                         (red ((dragon den)
                                                (viper nest)
                                                (fox hole)))
                                         (white ribbon)))))))))
  (add '() ral))

=>
assoc-list::add: named-object is NIL!
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod add (named-object (ral recursive-assoc-list) &optional ref)
```

28.5.262   recursive-assoc-list/add-empty-parcel

[recursive-assoc-list] [Methods]

DESCRIPTION:

Add a recursive-assoc-list object with NIL data (an empty level of
recursion) to the end of the top-level of a given recursive-assoc-list
object.

NB: Adding an empty parcel to a given recursive-assoc-list object will
    cause the method get-all-refs to fail on that recursive-assoc-list
    object.

ARGUMENTS:

- A recursive-assoc-list object.
- A symbol that will be the ID of the new, empty recursive-assoc-list
  object that is to be added.

OPTIONAL ARGUMENTS:

- <new-class> The name of an existing subclass of recursive-assoc-list that
the parcel should be promoted to.

RETURN VALUE:

A recursive-assoc-list object with DATA of NIL (the "empty parcel")

EXAMPLE:

```
;; Add two new empty parcels (the first a recursive-assoc-list, by default, the
;; second a rthm-seq-palette) and return the new list of REFS:
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
```

```
                                        (white ribbon)))))))))))
  (add-empty-parcel ral 'bricolage)
  (add-empty-parcel ral 'rsp 'rthm-seq-palette)
  (get-all-refs ral))
```

```
Mark set
=>
```

```
((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
 (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE)
 (BRICOLAGE) (RSP))
```

SYNOPSIS:

```
(defmethod add-empty-parcel ((ral recursive-assoc-list) id &optional new-class)
```

28.5.263   recursive-assoc-list/assoc-list-id-list

[recursive-assoc-list] [Functions]

DESCRIPTION:

Determine whether a given list contains only atoms which could be used as
assoc-list IDs. To pass the test, a given atom must be either a symbol, a
number or a string.

ARGUMENTS:

A list.

RETURN VALUE:

T or NIL indicating whether the atoms of the given list are all capable of
being used as assoc-list IDs. T = all can be used as assoc-list IDs.

EXAMPLE:

```
;; All of the elements in this list are either a symbol, a number or a
;; string. The list therefore returns a T when tested.
(let ((alil '(jim beam 3 "Allegro" 5 flute)))
  (assoc-list-id-list alil))
```

```
=> T
```

```
;; This list fails, as the last element is a list (and therefore not of type
```

```
;; string, number or symbol)
(let ((alil '(jim beam 3 "Allegro" 5 (flute))))
  (assoc-list-id-list alil))
```

```
=> NIL
```

SYNOPSIS:

```
(defun assoc-list-id-list (id-list)
```

28.5.264   recursive-assoc-list/ensemble

[recursive-assoc-list] [Classes]

NAME:

```
 ensemble

 File:            ensemble.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list ->
                  ensemble

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         Implementation of the ensemble class.

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   4th September 2001

 $$ Last modified:  17:12:22 Sat Sep 26 2020 CEST

 SVN ID: $Id$
```

28.5.265   ensemble/add-player

[ensemble] [Methods]

DESCRIPTION:

```
 Add a player to an existing ensemble. It will be added at the end of the
 list.
```

ARGUMENTS:

- The ensemble object.
- The new player, either as a player object or symbol. If the latter this
  becomes the id of the player we'll create.

OPTIONAL ARGUMENTS:

- The id of the instrument in the already existing instrument-palette.
  This is required if the player argument is a symbol. Default NIL.
- An instrument-palette object. Default =
  +slippery-chicken-standard-instrument-palette+.
- the midi-channel for the new player
- the microtones-midi-channel for the new player

RETURN VALUE:

 The player object added.

SYNOPSIS:

```
(defmethod add-player ((e ensemble) player
                       &optional
                         instrument-id
                         (instrument-palette
                          +slippery-chicken-standard-instrument-palette+)
                         ;; MDE Tue Jul 14 19:08:15 2020, Heidhausen
                         (midi-channel 1)
                         (microtones-midi-channel -1))
```

28.5.266   ensemble/balanced-load?

[ensemble] [Methods]

DATE:

 28th August 2014

DESCRIPTION:

 Determine whether the playing load is balanced across the players of the
 ensemble. By default, if the least active player is playing 80% of the time
 that the most active player is playing, we'll return T.

ARGUMENTS:

- an ensemble instance

OPTIONAL ARGUMENTS:

keyword arguments:
- :threshold. A number between 0.0 and 1.0 which represents the lowest
  ratio between the most and least active players.
- :stats-fun. One of the player methods which tots up statistics,
  i.e. total-notes, total-degrees, total-duration, or total-bars
- :ignore. A list of players (symbols) not to count in the sorting.

RETURN VALUE:

 T or NIL

SYNOPSIS:

```
(defmethod balanced-load? ((e ensemble) &key (threshold .8)
                                             (stats-fun #'total-duration)
                                             ignore)
```

28.5.267   ensemble/get-instrument

[ensemble] [Methods]

DATE:

 November 2nd 2018, Heidhausen

DESCRIPTION:

 Get the instrument object for a player in the ensemble. If the player
 doubles, then the optional <ins> argument is required and should be the ID
 of the instrument as defined in the ensemble's instrument palette
 (e.g. 'flute)

ARGUMENTS:

 - the ensemble object
 - the ID of a player in the ensemble

OPTIONAL ARGUMENTS:

 - the ID of an instrument if the given player doubles
 - T or NIL to issue a warning should the instrument not be found.

RETURN VALUE:

 The instrument object.

SYNOPSIS:

(defmethod get-instrument ((e ensemble) player &optional ins (warn t)))

28.5.268   ensemble/get-player

[ensemble] [Methods]

DESCRIPTION:

 Return a player object from an ensemble, if it exists.

ARGUMENTS:

 - An ensemble object.
 - The ID of a player.

RETURN VALUE:

 The player object or NIL if there's no such player.

SYNOPSIS:

(defmethod get-player ((e ensemble) player))

28.5.269   ensemble/get-players

[ensemble] [Methods]

DESCRIPTION:

 Return the IDs of the players from a given ensemble object.

ARGUMENTS:

 - An ensemble object.

RETURN VALUE:

 - A list of symbols that are the player IDs of the given ensemble object.

EXAMPLE:

```
(let ((ens (make-ensemble
            'ens
            '((flt ((flute piccolo) :midi-channel 1))
              (clr ((b-flat-clarinet)))
              (tpt ((b-flat-trumpet c-trumpet) :midi-channel 2))
              (vln ((violin))))
            :instrument-palette
            +slippery-chicken-standard-instrument-palette+)))
  (get-players ens))

=> (FLT CLR TPT VLN)
```

SYNOPSIS:

```
(defmethod get-players ((e ensemble))
```

28.5.270   ensemble/lotsa-combos

[ensemble] [Methods]

DATE:

 November 8th 2018, Heidhausen

DESCRIPTION:

 Get various permutations of the ensemble players, ranging from a single
 player up to one less than the number of players in the ensemble. This uses
 the shuffle method so is random, but as we use fixed-seed randomness we get
 repeatable results.

ARGUMENTS:

 - the ensemble object

OPTIONAL ARGUMENTS:

 - an integer to determine the number of times we permutate for any given
 number of players. Note that depending on the number of players in the
 ensemble and the number we're looking to put into a combo, we might not
 always be able to hit this target (e.g. there are only 6 possible
 permutations of three values). Default = 7.

RETURN VALUE:

 a list of lists of player IDs.

EXAMPLE:

```
(lotsa-combos (make-ensemble
                'ens
                '((flt ((flute piccolo)))
                  (clr ((b-flat-clarinet)))
                  (bsn bassoon)
                  (tpt ((b-flat-trumpet c-trumpet)))
                  (trb tenor-trombone)
                  (tb tuba)
                  (vln ((violin))))))
-->
((CLR FLT TRB BSN TB VLN) (TRB FLT CLR BSN TB TPT) (TB TRB FLT VLN TPT BSN)
 (CLR BSN TPT TB TRB FLT) (FLT TPT VLN BSN TRB TB) (FLT TPT BSN VLN TB CLR)
 (TPT FLT VLN CLR TB TRB) (BSN CLR TB TRB FLT) (TB FLT BSN VLN TPT)
 (FLT TPT BSN CLR TB) (FLT TRB CLR VLN TB) (BSN CLR VLN TPT TRB)
 (BSN TRB FLT TB CLR) (FLT CLR TPT TB VLN) (TRB CLR TPT TB) (FLT VLN TB TRB)
 (TRB VLN TB CLR) (TB VLN TRB BSN) (TPT TRB CLR BSN) (CLR FLT TRB VLN)
 (FLT TPT TRB CLR) (TPT CLR FLT) (TB TRB FLT) (TRB VLN FLT) (TB TPT CLR)
 (FLT BSN TPT) (FLT VLN BSN) (TB TPT TRB) (TPT FLT) (CLR TRB) (FLT CLR)
 (TRB CLR) (BSN CLR) (TRB TB) (BSN TPT) (FLT CLR BSN TPT TRB TB VLN) (FLT)
 (CLR) (BSN) (TPT) (TRB) (TB) (VLN))

(lotsa-combos (make-ensemble
                'ens
                '((flt ((flute piccolo)))
                  (clr ((b-flat-clarinet)))
                  (bsn bassoon)
                  (tpt ((b-flat-trumpet c-trumpet)))
                  (trb tenor-trombone)
                  (tb tuba)
                  (vln ((violin)))))
                10) ; <<--- get more combos
-->
((CLR TPT TB FLT TRB VLN) (BSN TRB VLN TB TPT CLR) (TB CLR FLT TRB BSN TPT)
 (TPT CLR BSN FLT VLN TRB) (VLN BSN TPT CLR TB TRB) (TRB BSN VLN TPT TB CLR)
 (TB TRB CLR VLN FLT BSN) (CLR TPT VLN TB FLT TRB) (TRB BSN VLN CLR FLT TPT)
 (TPT FLT CLR TB TRB VLN) (TPT TRB VLN BSN CLR) (BSN TB VLN TPT FLT)
 (BSN FLT TRB CLR TPT) (BSN TPT FLT CLR TB) (VLN TB TRB TPT FLT)
 (FLT TRB BSN TB VLN) (FLT CLR BSN TB TPT) (TRB FLT VLN TPT BSN)
 (BSN TPT TB TRB FLT) (TPT VLN BSN TRB TB) (BSN VLN TB CLR) (VLN CLR TB TRB)
 (CLR TB TRB FLT) (FLT BSN VLN TPT) (TPT BSN CLR TB) (TRB CLR VLN TB)
```

```
(CLR VLN TPT TRB) (TRB FLT TB CLR) (CLR TPT TB VLN) (TRB CLR TPT TB)
(VLN TB TRB) (VLN TB CLR) (VLN TRB BSN) (TRB CLR BSN) (FLT TRB VLN)
(TPT TRB CLR) (TPT CLR FLT) (TB TRB FLT) (TRB VLN FLT) (TB TPT CLR) (VLN BSN)
(TPT TRB) (TPT FLT) (CLR TRB) (FLT CLR) (TRB CLR) (BSN CLR) (TRB TB) (BSN TPT)
(FLT CLR BSN TPT TRB TB VLN) (FLT) (CLR) (BSN) (TPT) (TRB) (TB) (VLN))
```

SYNOPSIS:

```
(defmethod lotsa-combos ((e ensemble) &optional (try 7))
```

28.5.271   ensemble/make-ensemble

[ensemble] [Functions]

DESCRIPTION:

```
Make an ensemble object, specifying the players and associated
instruments.

NB: If you have an ensemble with a player doubling two instruments, be sure
    to indicate some keyword argument or other as
    (fl1 ((piccolo violin) :midi-channel 1)) works but
    (fl1 ((piccolo violin))) thinks that piccolo is a nested ensemble!!!
```

ARGUMENTS:

```
- An ID consisting of a symbol, string or number.
- A list of 2-element sublists that define the ensemble. See the above
  comment on adding a keyword argument for doubling players. An existing
  ensemble object can be passed here whereupon it will be cloned and the ID
  will be changed to the (new) given ID if it's not NIll. In this case
  however the keyword arguments will be ignored.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :instrument-palette. An instrument palette object. Default =
   +slippery-chicken-standard-instrument-palette+
- :bar-line-writers. Obsolete as no longer used.
```

RETURN VALUE:

```
An ensemble object.
```

EXAMPLE:

```
(let ((ens (make-ensemble
              'ens
              '((flt ((flute piccolo) :midi-channel 1))
                (clr ((b-flat-clarinet))))
              :instrument-palette
              +slippery-chicken-standard-instrument-palette+)))
  (print ens))

=>

ENSEMBLE: bar-line-writers: NIL
          players: (FLT CLR)
          (id instrument-palette): SLIPPERY-CHICKEN-STANDARD-INSTRUMENT-PALETTE
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 2
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: ENS, tag: NIL,
data: (
PLAYER: (id instrument-palette): SLIPPERY-CHICKEN-STANDARD-INSTRUMENT-PALETTE
doubles: T, cmn-staff-args: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: (FLT), next: (CLR)
NAMED-OBJECT: id: FLT, tag: NIL,
data:
[...]
data: (
INSTRUMENT: lowest-written:
[...]
NAMED-OBJECT: id: FLUTE, tag: NIL,
[...]
INSTRUMENT: lowest-written:
[...]
NAMED-OBJECT: id: PICCOLO, tag: NIL,
[...]
PLAYER: (id instrument-palette): SLIPPERY-CHICKEN-STANDARD-INSTRUMENT-PALETTE
doubles: NIL, cmn-staff-args: NIL
LINKED-NAMED-OBJECT: previous: (FLT), this: (CLR), next: NIL
NAMED-OBJECT: id: CLR, tag: NIL,
data:
INSTRUMENT: lowest-written:
[...]
NAMED-OBJECT: id: B-FLAT-CLARINET, tag: NIL,
```

)

SYNOPSIS:

```
(defun make-ensemble (id ensemble
                      &key bar-line-writers
                        (instrument-palette
                         +slippery-chicken-standard-instrument-palette+))
```

28.5.272   ensemble/num-notes

[ensemble] [Methods]

DESCRIPTION:

 Get the number of attacked notes in a given slippery-chicken object. This
 method accesses the ensemble object within the given slippery-chicken
 object to perform this task.

ARGUMENTS:

 - An ensemble object.

RETURN VALUE:

 An integer that is the total number of attacked notes in the given
 slippery-chicken object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) e e e e))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (num-notes (ensemble mini)))

=> 40
```

SYNOPSIS:

```
(defmethod num-notes ((e ensemble))
```

28.5.273   ensemble/num-players

[ensemble] [Methods]

DESCRIPTION:

Get the number of players in a given ensemble object.

ARGUMENTS:

- An ensemble object.

RETURN VALUE:

- An integer.

EXAMPLE:

```
(let ((ens (make-ensemble
            'ens
            '((flt ((flute piccolo) :midi-channel 1))
              (clr ((b-flat-clarinet)))
              (tpt ((b-flat-trumpet c-trumpet) :midi-channel 2))
              (vln ((violin))))
            :instrument-palette
            +slippery-chicken-standard-instrument-palette+)))
  (num-players ens))

=> 4
```

SYNOPSIS:

```
(defmethod num-players ((e ensemble))
```

28.5.274   ensemble/players-exist

[ensemble] [Methods]

DESCRIPTION:

 Produce an error message and drop into the debugger if the specified
 player IDs are not found within the given ensemble object.

ARGUMENTS:

 - An ensemble object.
 - A list of symbols that are the IDs of the players sought.

RETURN VALUE:

 NIL if the specified player ID is present within the given ensemble object,
 otherwise drops into the debugger with an error.

EXAMPLE:

```
;;; Returns NIL if a player with the specified ID is found in the given
;;; ensemble object.
(let ((ens (make-ensemble
             'ens
             '((flt ((flute piccolo) :midi-channel 1))
               (clr ((b-flat-clarinet)))
               (tpt ((b-flat-trumpet c-trumpet) :midi-channel 2))
               (vln ((violin))))
             :instrument-palette
             +slippery-chicken-standard-instrument-palette+)))
  (players-exist ens '(vln)))

=> NIL

;; Drops into the debugger with an error if no player with the specified ID is
;; found in the given ensemble object.
(let ((ens (make-ensemble
             'ens
             '((flt ((flute piccolo) :midi-channel 1))
               (clr ((b-flat-clarinet)))
               (tpt ((b-flat-trumpet c-trumpet) :midi-channel 2))
               (vln ((violin))))
             :instrument-palette
             +slippery-chicken-standard-instrument-palette+)))
  (players-exist ens '(vla)))

=>
ensemble::players-exist: VLA is not a member of the ensemble
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod players-exist ((e ensemble) players)
```

28.5.275   ensemble/sort-players

[ensemble] [Methods]

DATE:

 27th August 2013

DESCRIPTION:

 Return a hierarchical list of players sorted by e.g. how much they're
 playing.

ARGUMENTS:

 - An ensemble object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :stats-fun. One of the player methods which tots up statistics,
    i.e. total-notes, total-degrees, total-duration, or total-bars
 - :ignore. A list of players (symbols) not to count in the sorting.

RETURN VALUE:

 A list of all the players in the ensemble ordered by its statistics.

SYNOPSIS:

```
(defmethod sort-players ((e ensemble) &key (stats-fun #'total-duration)
                                           ignore)
```

28.5.276   ensemble/tessitura

[ensemble] [Methods]

DESCRIPTION:

 Get the average pitch of a given slippery-chicken object. This method
 accesses the ensemble object within the given slippery-chicken object to
 perform this task.

NB: This method processes data in relationship to degrees of the current
    tuning system (scale), which is quarter-tone by default. It is
    therefore possible, when generating a piece using only chromatic
    pitches but within a non-chromatic tuning to get microtonal results.

ARGUMENTS:

 - An ensemble object.

RETURN VALUE:

 An integer that is the average pitch of the given slippery-chicken object
 in degrees.

EXAMPLE:

```
;;; Change the tuning to chromatic first to get an accurate result:
(in-scale :chromatic)

=> #<tuning "chromatic-scale">

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) e e e e))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (tessitura (ensemble mini)))

=> C4
```

SYNOPSIS:

```
(defmethod tessitura ((e ensemble))
```

28.5.277   recursive-assoc-list/get-all-refs

[recursive-assoc-list] [Methods]

DESCRIPTION:

Return a list of all the keys (REFS) in a given recursive-assoc-list
object. Nested keys are given in FULL-REF form, i.e. a list that is the
path of keys to the specific key.

Keys that are not part of nesting-path are also returned as lists
(single-item lists) by default. An optional argument allows these to be
returned as individual symbols rather than lists.

NB This will only work on the top-level object due to the creation of
   references when linking. If you're interested in the full refs of a
   sub-ral, try ths get-this-refs method.

ARGUMENTS:

 - A recursive-assoc-list object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to return single REFS (non-nested keys) as
   lists or as individual symbols. T = as list. Default = T.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; By default all keys are returned as lists, even single (non-nested) keys
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-all-refs ral))

=> ((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
    (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE))

;; Setting the optional argument to NIL returns non-nested keys as symbols
;; rather than lists
(let ((ral (make-ral 'mixed-bag
```

```
                        '((jim beam)
                          (wild turkey)
                          (four ((roses red)
                                 (violets ((blue velvet)
                                           (red ((dragon den)
                                                 (viper nest)
                                                 (fox hole)))
                                           (white ribbon)))))))))
  (get-all-refs ral nil))

=> (JIM WILD (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
    (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE))
```

SYNOPSIS:

```
(defmethod get-all-refs ((ral recursive-assoc-list)
                         &optional
                         (single-ref-as-list t))
```

28.5.278   recursive-assoc-list/get-data

[recursive-assoc-list] [Methods]

DESCRIPTION:

Return the named-object (or linked-named-object) that is identified by a
specified key within a given recursive-assoc-list object.

NB: This method returns the named object itself, not just the data
    associated with the key (use assoc-list::get-data-data for that).

ARGUMENTS:

 - A symbol that is the key (id) of the named-object sought, or a list of
   symbols that are the path to the desired named-object within the given
   recursive-assoc-list.
 - The recursive-assoc-list object in which it is sought.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether a warning is printed if the specified key
   cannot be found within the given assoc-list. T = print. Default = T.

RETURN VALUE:

A named-object is returned if the specified key is found within the given
recursive-assoc-list object.

NIL is returned and a warning is printed if the specified key is not found
in the given recursive-assoc-list object. This applies, too, when a nested
key is specified without including the other keys that are the path to that
key (see example).

EXAMPLE:

```
;; Get a named-object from the top-level of the recursive-assoc-list object
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                       (white ribbon)))))))))
  (get-data 'wild ral))

=>
NAMED-OBJECT: id: WILD, tag: NIL,
data: TURKEY

;; A list including all keys that are the path to the specified key is required
;; to get nested named-objects
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                       (white ribbon)))))))))
  (get-data '(four violets white) ral))

=>
NAMED-OBJECT: id: WHITE, tag: NIL,
data: RIBBON

;; Searching for a key that is not present in the given recursive-assoc-list
;; object returns NIL and a warning
(let ((ral (make-ral 'mixed-bag
```

```
                               '((jim beam)
                                 (wild turkey)
                                 (four ((roses red)
                                        (violets ((blue velvet)
                                                  (red ((dragon den)
                                                        (viper nest)
                                                        (fox hole)))
                                                  (white ribbon)))))))))
  (get-data 'johnnie ral))

=> NIL
WARNING:
   assoc-list::get-data: Could not find data with key JOHNNIE
   in assoc-list with id MIXED-BAG

;; Searching for a nested key without specifying the path to that key within a
;; list also returns a NIL and a warning
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-data 'fox ral))

=> NIL
WARNING:
   assoc-list::get-data: Could not find data with key FOX
   in assoc-list with id MIXED-BAG
```

SYNOPSIS:

```
(defmethod get-data :around (ids (ral recursive-assoc-list)
                                 &optional (warn t))
```

28.5.279   recursive-assoc-list/get-first

[recursive-assoc-list] [Methods]

DESCRIPTION:

```
 Returns the first named-object in the DATA slot of the given
 recursive-assoc-list object.
```

ARGUMENTS:

- A recursive-assoc-list object.

RETURN VALUE:

A named-object that is the first object in the DATA slot of the given
recursive-assoc-list object.

EXAMPLE:

```
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-first ral))

=>
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM
```

SYNOPSIS:

```
(defmethod get-first ((ral recursive-assoc-list))
```

28.5.280   recursive-assoc-list/get-first-ref

[recursive-assoc-list] [Methods]

DESCRIPTION:

Get the full reference into the given recursive-assoc-list object of the
first named-object in the given recursive-assoc-list object.

NB: If the <ral> argument happens to be a recursive-assoc-list object that
    is contained within another recursive-assoc-list object (i.e. is a
    nested recursive-assoc-list object), then the result is the reference
    into the top-level recursive-assoc-list object, not the argument.

ARGUMENTS:

- A recursive-assoc-list object.

RETURN VALUE: EXAMPLE:

```
;; A simple call returns the first top-level named-object
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-first-ref ral))

=> (JIM)

;; Return the first ref of a nested recursive-assoc-list object
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-first-ref (get-data-data '(four violets) ral)))

=> (FOUR VIOLETS BLUE)
```

SYNOPSIS:

```
(defmethod get-first-ref ((ral recursive-assoc-list))
```

28.5.281   recursive-assoc-list/get-last

[recursive-assoc-list] [Methods]

DESCRIPTION:

Get the last named-object in a given recursive-assoc-list object.

NB: This method functions linearly, not hierarchically. The last named

object is therefore not necessarily the deepest of a nest, but the last
listed.

ARGUMENTS:

 - A recursive-assoc-list object.

RETURN VALUE:

 A named-object (or linked-named-object).

EXAMPLE:

```
;; This returns '(white ribbon), not '(fox hole)
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-last ral))

=>
NAMED-OBJECT: id: WHITE, tag: NIL,
data: RIBBON
```

SYNOPSIS:

```
(defmethod get-last ((ral recursive-assoc-list))
```

28.5.282   recursive-assoc-list/get-last-ref

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Get the last REF (path of nested keys) of the given recursive-assoc-list
 object.

 NB: This method functions linearly, not hierarchically. The last-ref may
     not be the deepest nested.

ARGUMENTS:

 - A recursive-assoc-list object.

RETURN VALUE:

 Returns a list that is the last REF of the given recursive-assoc-list
 object.

EXAMPLE:

```
;; Typical usage with nesting
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-last-ref ral))

=> (FOUR VIOLETS WHITE)

;; Returns the last-ref as a list even if the given recursive-assoc-list object
;; contains no nesting
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four roses)))))
  (get-last-ref ral))

=> (FOUR)
```

SYNOPSIS:

```
(defmethod get-last-ref ((ral recursive-assoc-list))
```

28.5.283   recursive-assoc-list/get-previous

[recursive-assoc-list] [Methods]

DESCRIPTION:

Get the previous named-object in the given recursive-assoc-list object by
specifying the ID of a named-object contained within that given
recursive-assoc-list object.

An optional argument allows for the retrieval of a previous named-object
that is more than one step back in the given recursive-assoc-list object
(i.e., not the named-object that immediately precedes the specified key).

If the number given for the optional <how-many> argument is greater than
the number of items in the given recursive-assoc-list object, the value
returned will be a negative number.

The method proceeds linearly, not hierarchically, when getting previous
named-objects from further down into nested assoc-lists. In other words,
the named-object immediately previous to (white ribbon) in this nested list
is (fox hole), which is at a deeper level, not (red ...) or (blue velvet),
which are at the same level:
((blue velvet)
 (red ((dragon den)
       (viper nest)
       (fox hole)))
 (white ribbon)

In order to retrieve objects that are nested more deeply, the list that is
the <keys> argument must consist of the consecutive path of keys leading to
that object. If only the key of a named object that is deeper in the list
is given, and not the path of keys to that object, a warning will be
printed that the given key cannot be found in the list.

NB: When this method is applied to keys that contain further assoc-list
    objects, the method will drop into the debugger with an error.

ARGUMENTS:

 - A recursive-assoc-list object.
 - A list containing one or more symbols that are either the ID of the
   specified named object or the path of keys to that object within the
   given recursive-assoc-list object.

OPTIONAL ARGUMENTS:

 - An integer indicating how many steps back in the given
   recursive-assoc-list from the specified named-object to look when
   retrieving the desired object (e.g. 1 = immediately previous object, 2 =
   the one before that etc.)

RETURN VALUE:

 A linked-named-object.

EXAMPLE:

```
;; Get the object immediately previous to that with the key WILD returns the
;; object with key JIM and data BEAM
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-previous ral '(wild)))

=>
LINKED-NAMED-OBJECT: previous: NIL, this: (JIM), next: (WILD)
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM

;; Attempting to get the previous object from the key FOUR, which contains a
;; nested list, returns an error unless the first key in the nested list is
;; also included
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-previous ral '(four)))

=>
There is no applicable method for the generic function
  #<STANDARD-GENERIC-FUNCTION PREVIOUS (1)>
when called with arguments
  (
NAMED-OBJECT: id: FOUR, tag: NIL,

(let ((ral (make-ral 'mixed-bag
```

```
                               '((jim beam)
                                 (wild turkey)
                                 (four ((roses red)
                                        (violets ((blue velvet)
                                                  (red ((dragon den)
                                                        (viper nest)
                                                        (fox hole)))
                                                  (white ribbon)))))))))
  (get-previous ral '(four roses)))

=>
LINKED-NAMED-OBJECT: previous: (JIM), this: (WILD), next: (FOUR ROSES)
NAMED-OBJECT: id: WILD, tag: NIL,
data: TURKEY

;; The method defines the previous object linearly, not hierarchically; i.e.,
;; the previous object to (white ribbon) here is (fox hole) and not (red ...)
(let ((ral (make-ral 'mixed-bag
                       '((jim beam)
                         (wild turkey)
                         (four ((roses red)
                                (violets ((blue velvet)
                                          (red ((dragon den)
                                                (viper nest)
                                                (fox hole)))
                                          (white ribbon)))))))))
  (get-previous ral '(four violets white)))

=>
LINKED-NAMED-OBJECT: previous: (FOUR VIOLETS RED VIPER),
this: (FOUR VIOLETS RED FOX),
next: (FOUR VIOLETS WHITE)
NAMED-OBJECT: id: FOX, tag: NIL,
data: HOLE

;; Use the <how-many> argument to retrieve previous objects further back than
;; the immediate predecessor
(let ((ral (make-ral 'mixed-bag
                       '((jim beam)
                         (wild turkey)
                         (four ((roses red)
                                (violets ((blue velvet)
                                          (red ((dragon den)
                                                (viper nest)
                                                (fox hole)))
                                          (white ribbon)))))))))
```

```
  (get-previous ral '(four violets white) 4))

=>
LINKED-NAMED-OBJECT: previous: (FOUR ROSES),
this: (FOUR VIOLETS BLUE),
next: (FOUR VIOLETS RED DRAGON)
NAMED-OBJECT: id: BLUE, tag: NIL,
data: VELVET

;; Using a <how-many> value greater than the number of items in the given
;; recursive-assoc-list object returns a negative number
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (get-previous ral '(four violets white) 14))

=> -7
```

SYNOPSIS:

```
(defmethod get-previous ((ral recursive-assoc-list) keys
                         &optional (how-many 1))
```

28.5.284   recursive-assoc-list/get-this-refs

[recursive-assoc-list] [Methods]

DATE:

 5th July 2019, Essen Werden

DESCRIPTION:

 Get the <this> slots of each named-object (or subclass) in a
 recursive-assoc-list (ral). This should work on top-level rals and sub-rals
 alike, though rals with only sub-rals probably won't return anything. Unlike
 get-all-refs, this doesn't traverse recursive objects.

ARGUMENTS:

– the recursive-assoc-list object

RETURN VALUE:

a list of references (most probably simple lists)

SYNOPSIS:

```
(defmethod get-this-refs ((ral recursive-assoc-list))
```

28.5.285 recursive-assoc-list/link-named-objects

[recursive-assoc-list] [Methods]

DESCRIPTION:

Create linked-named-objects from the named-objects in the data slots of the
given recursive-assoc-list object. The linked-named-objects created hold
keys that serve as pointers to the previous and next objects in the given
recursive-assoc-list object, whether recursive or not.

The optional <previous> and <higher-next> arguments are only for internal
recursive calls and so shouldn't be given by the user.

ARGUMENTS:

– A recursive-assoc-list object.

OPTIONAL ARGUMENTS:

– <previous>
– <higher-next>

EXAMPLE:

```
;;; The recursive-assoc-list may not be linked on creation, evident here
;;; through the value of the LINKED slot
(make-ral 'mixed-bag
               '((jim beam)
                 (wild turkey)
                 (four ((roses red)
                        (violets ((blue velvet)
                                  (red ((dragon den)
                                        (viper nest)
```

```
                                                     (fox hole)))
                                             (white ribbon)))))))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: NIL,
data: (
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM

NAMED-OBJECT: id: WILD, tag: NIL,
data: TURKEY
[...]

;; The recursive-assoc-list object and the named-objects it contains are linked
;; after applying the link-named-objects method
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (link-named-objects ral))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: NIL,
data: (
LINKED-NAMED-OBJECT: previous: NIL, this: (JIM), next: (WILD)
```

```
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM

LINKED-NAMED-OBJECT: previous: (JIM), this: (WILD), next: (FOUR ROSES)
NAMED-OBJECT: id: WILD, tag: NIL,
data: TURKEY
```

RETURN VALUE:

 the recursive-assoc-list object

SYNOPSIS:

```
(defmethod link-named-objects ((ral recursive-assoc-list)
                                 &optional previous higher-next)
```

28.5.286   recursive-assoc-list/lisp-assoc-listp

[recursive-assoc-list] [Functions]

DESCRIPTION:

 Determine whether a given list can has the structure of a lisp
 assoc-list. This is assessed based on each of the elements being a 2-item
 list, of which the first is a symbol, number or string (qualifies as a
 key).

 The optional argument <recurse-simple-data> allows the data portion of
 key/data pairs to be viewed as flat lists rather than as recursive lists.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether to consider lists of 2-item lists in the data
 position of a given key/data pair to be a list or a recursive list.
 T = list. Default = T.

RETURN VALUE:

 T or NIL. T = the tested list can be considered a Lisp assoc-list.

EXAMPLE:

```
;; A list of 2-item lists, each of whose item are all either a symbol, number,
;; or string, can be considered a Lisp assoc-list.
(let ((lal '((roses red) (3 "allegro") (5 flute))))
  (lisp-assoc-listp lal))

=> T

;; By default, lists of 2-item lists in the DATA portion of a key/data pair
;; will be considered as a simple list, rather than a recursive list, resulting
;; in the tested list passing as T.
(let ((lal '((1 2) (3 ((4 5) (6 7))) (8 9))))
  (lisp-assoc-listp lal))

=> T

;; Setting the optional argument to NIL will cause the same list to fail with
(let ((lal '((1 2) (3 ((4 5) (6 7))) (8 9))))
  (lisp-assoc-listp lal nil))

=> NIL
```

SYNOPSIS:

```
(defun lisp-assoc-listp (candidate &optional (recurse-simple-data t))
```

28.5.287 recursive-assoc-list/make-ral

[recursive-assoc-list] [Functions]

DESCRIPTION:

Create a recursive-assoc-list object, which allows and automatically
instantiates association lists inside of association lists to any level of
nesting.

ARGUMENTS:

- A symbol that is the object's ID.
- A list of nested lists, or a list.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :recurse-simple-data. T or NIL to indicate whether to recursively
  instantiate a recursive-assoc-list in place of data that appears to be a
```

   simple assoc-list (i.e. a 2-element list). If NIL, the data of 2-element
   lists whose second element is a number or a symbol will be ignored,
   therefore remaining as a list. For example, this data would normally
   result in a recursive call: (y ((2 23) (7 28) (18 2))). T = replace
   assoc-list data with recursive-assoc-lists. Default = T.
 - :full-ref. Nil or a list representing the path to a nested
   recursive-assoc-list object within the given recursive-assoc-list object,
   starting from the top level of the given object. When NIL, the given
   recursive-assoc-list object itself is the top level. Default = NIL.
 - :tag. A symbol that is another name, description etc. for the given
   recursive-assoc-list object. The tag may be used for identification but
   not for searching purposes. Default = NIL.
 - :warn-not-found. T or NIL to indicate whether a warning is printed when
   an index which doesn't exist is used for look-up. Default = T.

RETURN VALUE:

 Returns a recursive-assoc-list object.

EXAMPLE:

```
;; Create a recursive-assoc-list object with default keyword argument values
(make-ral 'mixed-bag
          '((jim beam)
            (wild turkey)
            (four ((roses red)
                   (violets ((blue velvet)
                             (red ((dragon den)
                                   (viper nest)
                                   (fox hole)))
                             (white ribbon)))))))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: NIL,
data: (
[...]

;; Use the class's get-all-refs method to show that by default, simple data is
```

```
;; recursed. The sublists in the second list in this example are processed as
;; nested lists
(let ((ral (make-ral 'ral-test
                     '((1 one)
                       (2 ((3 4) (5 6)))
                       (3 three)))))
  (get-all-refs ral))

=> ((1) (2 3) (2 5) (3))

;; Using the same data, but setting the :recurse-simple-data argument to NIL
;; will cause the method to process simple data as a unit rather than nested
;; lists
(let ((ral (make-ral 'ral-test
                     '((1 one)
                       (2 ((3 4) (5 6)))
                       (3 three))
                     :recurse-simple-data nil)))
  (get-all-refs ral))

=> ((1) (2) (3))
```

SYNOPSIS:

```
(defun make-ral (id ral &key (recurse-simple-data t) (warn-not-found t)
                 (tag nil) (full-ref nil))
```

28.5.288   recursive-assoc-list/palette

[recursive-assoc-list] [Classes]

NAME:

 palette

File:	palette.lsp
Class Hierarchy:	named-object -> linked-named-object -> sclist -> circular-sclist -> assoc-list -> recursive-assoc-list -> palette
Version:	1.0.11
Project:	slippery chicken (algorithmic composition)

```
Purpose:            Implementation of the palette class which adds very little
                    to its direct superclass assoc-list (as of yet) but spawns
                    a new base type for more specialised palettes.

Author:             Michael Edwards: m@michael-edwards.org

Creation date:      19th February 2001

$$ Last modified: 19:06:49 Sat May  7 2016 WEST

SVN ID: $Id$
```

28.5.289   palette/instrument-palette

[palette] [Classes]

NAME:

```
instrument-palette

File:               instrument-palette.lsp

Class Hierarchy:    named-object -> linked-named-object -> sclist ->
                    circular-sclist -> assoc-list -> recursive-assoc-list ->
                    palette -> instrument-palette

Version:            1.0.11

Project:            slippery chicken (algorithmic composition)

Purpose:            Implementation of the instrument-palette class which
                    intantiates instruments to be used in an ensemble
                    instance.

Author:             Michael Edwards: m@michael-edwards.org

Creation date:      6th September 2001

$$ Last modified:  12:22:46 Thu Sep 20 2018 CEST

SVN ID: $Id$
```

28.5.290   instrument-palette/auto-set-subset-id

[instrument-palette] [Methods]

DATE:

 July 27th 2018, Heidhausen

DESCRIPTION:

 Set the subset-id slot to be the same as the id slot for each instrument in
 the palette.

ARGUMENTS:

 - an instrument-palette instance

RETURN VALUE:

 - the instrument instance

SYNOPSIS:

(defmethod auto-set-subset-id ((ip instrument-palette))

28.5.291   instrument-palette/make-instrument-palette

[instrument-palette] [Functions]

DESCRIPTION:

 Create an instrument-palette object from a list of instrument descriptions
 based on the keyword arguments of make-instrument.

ARGUMENTS:

 - A symbol that will serve as the ID for the instrument-palette object.
 - A list of instrument descriptions based on the keyword arguments of
   make-instrument.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :warn-not-found. T or NIL to indicate whether a warning is printed when
   an index which doesn't exist is used for look-up. Default = T.

RETURN VALUE:

An instrument palette.

EXAMPLE:

```
;; Returns an instrument-palette object
(make-instrument-palette 'inst-pal
                         '((piccolo (:transposition-semitones 12
                                     :lowest-written d4 :highest-written c6))
                           (bf-clarinet (:transposition-semitones -2
                                         :lowest-written e3
                                         :highest-written c6))
                           (horn (:transposition f :transposition-semitones -7
                                  :lowest-written f2 :highest-written c5))
                           (violin (:lowest-written g3 :highest-written c7
                                    :chords t))
                           (viola (:lowest-written c3 :highest-written f6
                                   :chords t))))

=>
INSTRUMENT-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 5
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 5, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: INST-PAL, tag: NIL,
data: (
[...]
```

SYNOPSIS:

```
(defun make-instrument-palette (id ip &key (warn-not-found t))
```

28.5.292   instrument-palette/set-prefers-high

[instrument-palette] [Methods]

DATE:

05 Feb 2011

DESCRIPTION:

Set the PREFERS-NOTES slot of a specified instrument object within a given
instrument-palette object to 'HIGH. The instrument object is specified
using the ID symbol assigned to it within the instrument-palette object
definition.

NB: The optional argument is actually required, but is listed as optional
because of the attributes of the instrument class method.

ARGUMENTS:

- An instrument-palette object.

OPTIONAL ARGUMENTS:

- A symbol that is the ID of the instrument object within the
  instrument-palette object definition.

RETURN VALUE:

Returns the symbol 'HIGH.

EXAMPLE:

```
;; Define an instrument-palette object, then set the PREFERS-NOTES slot of the
;; instrument object 'piccolo within that instrument-palette object to 'HIGH
(let ((ip (make-instrument-palette 'inst-pal
                                   '((piccolo (:transposition-semitones 12
                                               :lowest-written d4
                                               :highest-written c6))
                                     (bf-clarinet (:transposition-semitones -2
                                                   :lowest-written e3
                                                   :highest-written c6))
                                     (horn (:transposition f
                                            :transposition-semitones -7
                                            :lowest-written f2
                                            :highest-written c5))
                                     (violin (:lowest-written g3
                                              :highest-written c7
                                              :chords t))
                                     (viola (:lowest-written c3
                                             :highest-written f6
                                             :chords t))))))
  (set-prefers-high ip 'piccolo))

=> HIGH
```

SYNOPSIS:

```
(defmethod set-prefers-high ((ip instrument-palette) &optional instrument)
```

28.5.293   instrument-palette/set-prefers-low

[instrument-palette] [Methods]

DATE:

 05 Feb 2011

DESCRIPTION:

 Set the PREFERS-NOTES slot of a specified instrument object within a given
 instrument-palette object to 'LOW. The instrument object is specified
 using the ID symbol assigned to it within the instrument-palette object
 definition.

 NB: The optional argument is actually required, but is listed as optional
     because of the attributes of the instrument class method.

ARGUMENTS:

 - An instrument-palette object.

OPTIONAL ARGUMENTS:

 - A symbol that is the ID of the instrument object within the
   instrument-palette object definition.

RETURN VALUE:

 Returns the symbol 'LOW.

EXAMPLE:

```
;; Define an instrument-palette object, then set the PREFERS-NOTES slot of the
;; instrument object 'piccolo within that instrument-palette object to 'LOW
(let ((ip (make-instrument-palette 'inst-pal
                                  '((piccolo (:transposition-semitones 12
                                              :lowest-written d4
                                              :highest-written c6))
                                    (bf-clarinet (:transposition-semitones -2
```

```
                                           :lowest-written e3
                                           :highest-written c6))
                              (horn (:transposition f
                                     :transposition-semitones -7
                                     :lowest-written f2
                                     :highest-written c5))
                              (violin (:lowest-written g3
                                        :highest-written c7
                                        :chords t))
                              (viola (:lowest-written c3
                                       :highest-written f6
                                       :chords t))))))
  (set-prefers-low ip 'piccolo))

=> LOW
```

SYNOPSIS:

```
(defmethod set-prefers-low ((ip instrument-palette) &optional instrument)
```

28.5.294   palette/pitch-seq-palette

[palette] [Classes]

NAME:

```
 pitch-seq-palette

 File:            pitch-seq-palette.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list ->
                  palette -> pitch-seq-palette

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         Implementation of the pitch-seq-palette class.

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   19th February 2001

 $$ Last modified:  14:01:06 Sat Jul 11 2020 CEST
```

SVN ID: Id

28.5.295   pitch-seq-palette/add-inversions

[pitch-seq-palette] [Methods]

DESCRIPTION:

Add inversions of the existing pitch-seq objects in a given
pitch-seq-palette object to the end of that pitch-seq-palette object. (See
pitch-seq::invert for more details on slippery-chicken inversions.)

ARGUMENTS:

- A pitch-seq-palette object.

RETURN VALUE:

Always returns T.

EXAMPLE:

```
;; Create a pitch-seq-palette object and print the DATA of the pitch-seq
;; objects it contains; then apply the add-inversions method and print the same
;; DATA to see the changes
(let ((mpsp (make-psp 'mpsp 5 '((2 5 3 1 4)
                                (1 4 2 5 3)
                                (5 1 3 2 4)
                                (2 3 4 5 1)
                                (3 2 4 1 5)))))
  (print (loop for ps in (data mpsp)
            collect (data ps)))
  (add-inversions mpsp)
  (print (loop for ps in (data mpsp)
            collect (data ps))))

=>
((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4) (2 3 4 5 1) (3 2 4 1 5))

((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4) (2 3 4 5 1) (3 2 4 1 5) (4 1 3 5 2)
 (5 2 4 1 3) (1 5 3 4 2) (4 3 2 1 5) (3 4 2 5 1))
```

SYNOPSIS:

```
(defmethod add-inversions ((psp pitch-seq-palette))
```

28.5.296   pitch-seq-palette/combine

[pitch-seq-palette] [Methods]

DESCRIPTION:

Create a new pitch-seq-palette object by combining the pitch-seq lists from
one pitch-seq-palette object with those of another.

The method combines the contents of the two given rthm-seq-palette objects
consecutively; i.e., the first pitch-seq object of the first
pitch-seq-palette is combined with the first pitch-seq object of the other,
then the second with the second, the third with the third etc.

If one pitch-seq-palette object contains more pitch-seq objects than the
other, the method cycles through the shorter one until all of the members
of the longer one have been handled. The new pitch-seq-palette object will
therefore contain the same number of pitch-seq objects as is in the longest
of the two starting pitch-seq-palette objects.

It is not necessary for the lengths of the pitch-seq objects in the two
starting pitch-seq-palette objects to be the same.

NB Both pitch-seq-palettes are reset by this method.

ARGUMENTS:

 - A first pitch-seq-palette object.
 - A second pitch-seq-palette object.

RETURN VALUE:

 A pitch-seq-palette object.

EXAMPLE:

```
;;; Combine two pitch-seq-palette objects of the same length, each of whose
;;; pitch-seqs are the same length
(let ((mpsp1 (make-psp 'mpsp1 5 '((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4))))
      (mpsp2 (make-psp 'mpsp2 5 '((2 3 4 5 1) (3 2 4 1 5) (3 2 1 5 4)))))
  (combine mpsp1 mpsp2))

=>
PITCH-SEQ-PALETTE: num-notes: 10, instruments: NIL
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
```

```
                            num-data: 3
                            linked: NIL
                            full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "MPSP1-MPSP2", tag: NIL,
data: (
PITCH-SEQ: notes: NIL
[...]
data: (2 5 3 1 4 2 3 4 5 1)
PITCH-SEQ: notes: NIL
[...]
data: (1 4 2 5 3 3 2 4 1 5)
[...]
PITCH-SEQ: notes: NIL
[...]
data: (5 1 3 2 4 3 2 1 5 4)
)


;; When combining pitch-seq-palette objects of different lengths, the method
;; cyles through the shorter of the two
(let ((mpsp1 (make-psp 'mpsp1 5 '((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4))))
      (mpsp2 (make-psp 'mpsp2 5 '((2 3 4 5 1) (3 2 4 1 5)))))
  (loop for ps in (data (combine mpsp1 mpsp2))
        collect (data ps)))

=> ((2 5 3 1 4 2 3 4 5 1) (1 4 2 5 3 3 2 4 1 5) (5 1 3 2 4 2 3 4 5 1))

;; The two starting pitch-seq-palette objects are not required to have
;; pitch-seq objects of the same length
(let ((mpsp1 (make-psp 'mpsp1 5 '((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4))))
      (mpsp2 (make-psp 'mpsp2 3 '((2 3 4) (3 2 4)))))
  (loop for ps in (data (combine mpsp1 mpsp2))
        collect (data ps)))

=> ((2 5 3 1 4 2 3 4) (1 4 2 5 3 3 2 4) (5 1 3 2 4 2 3 4))
```

SYNOPSIS:

```
(defmethod combine ((psp1 pitch-seq-palette) (psp2 pitch-seq-palette))
```

28.5.297   pitch-seq-palette/make-psp

[pitch-seq-palette] [Functions]

DESCRIPTION:

 Create a pitch-seq-palette object from an ID, a specified number of notes,
 and a list of lists of numbers representing the pitch curve of the intended
 pitch-seq objects.

ARGUMENTS:

 - A symbol that is to be the ID of the pitch-seq-palette to be created.
 - An integer that is the number of notes there are to be in each pitch-seq
   object created.
 - A list of lists, each of which contained list is a list of numbers
   representing the pitch curve of the intended pitch-seq object.

RETURN VALUE:

 A pitch-seq-palette object.

EXAMPLE:

```
;; Returns a pitch-seq-palette object
(make-psp 'mpsp 5 '((2 5 3 1 4)
                    (1 4 2 5 3)
                    (5 1 3 2 4)
                    (2 3 4 5 1)
                    (3 2 4 1 5)))

=>
PITCH-SEQ-PALETTE: num-notes: 5, instruments: NIL
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 5
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 5, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MPSP, tag: NIL,
data: (
PITCH-SEQ: notes: NIL
[...]
```

```
data: (2 5 3 1 4)
PITCH-SEQ: notes: NIL
[...]
data: (1 4 2 5 3)
PITCH-SEQ: notes: NIL
[...]
data: (5 1 3 2 4)
PITCH-SEQ: notes: NIL
[...]
data: (2 3 4 5 1)
PITCH-SEQ: notes: NIL
[...]
data: (3 2 4 1 5)
)

;; Interrupts with an error if any of the <pitch-seqs> lists is not of the
;; length specified by <num-notes>
(make-psp 'mpsp 5 '((1 2 1 1 3)
                    (1 3 2 1 5)
                    (1 3 5 6 7 8)))

=>
pitch-seq-palette::verify-and-store:
In pitch-seq MPSP-ps-3 from palette MPSP:
Each pitch sequence must have 5 notes (you have 6):
[...]
 (1 3 5 6 7 8))
   [Condition of type SIMPLE-ERROR]

;; We can also assign instruments to specific pitch-seqs by passing their names
;; (not player names, but instrument names) e.g.
(make-psp 'mpsp 5 '((1 2 1 1 3)
                    ((1 3 2 1 5) violin flute)
                    (1 3 5 6 7)))
```

SYNOPSIS:

```
(defun make-psp (id num-notes pitch-seqs)
```

28.5.298   palette/rthm-seq-palette

[palette] [Classes]

NAME:

```
 set-palette
```

```
File:              rthm-seq-palette.lsp

Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list -> recursive-assoc-list ->
                   palette -> rthm-seq-palette

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Implementation of the rthm-seq-palette class.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     19th February 2001

$$ Last modified:  15:49:30 Tue Oct 27 2020 CET

SVN ID: $Id$
```

28.5.299   rthm-seq-palette/chop

[rthm-seq-palette] [Methods]

DESCRIPTION:

Applies the chop method to each rthm-seq object in the given
rthm-seq-palette object (see rthm-seq-bar::chop for details). Returns a
new rthm-seq-palette object with the same structure as the argument, but
with a further level of nesting: Each rthm-seq object in the argument is
replaced by a list of rthm-seq objects that are each one "slice" of the
original rthm-seq objects.

The chop method is the basis for slippery-chicken's feature of
intra-phrasal looping.

NB: Since the chop method functions by comparing each beat of a given
    rthm-seq-bar object to the specified <chop-points> pattern for
    segmenting that beat, all rthm-seq-bar objects in the given
    rthm-seq-palette object must be evenly divisible by the beat for which
    the pattern is defined. For example, if the <chop-points> argument
    defines a quarter-note, all bars in the given rthm-seq-palette object
    must be evenly divisible by a quarter-note, and a rthm-seq-palette
    consisting of a rthm-seq object with a 2/4, a 3/4 and a 3/8 bar would
    fail at the 3/8 bar with an error.

NB: The <unit> argument must be a duplet rhythmic value (i.e. 32, 's, 'e
    etc.) and cannot be a tuplet value (i.e. 'te 'fe etc.).

NB: In order for the resulting chopped rhythms to be parsable by LilyPond
    and CMN, there can be no tuplets (triplets etc.) among the rhythms to
    be chopped. Such rhythms will result in LilyPond and CMN errors. This
    has only minimal bearing on any MIDI files produced, however, and these
    can potentially be imported into notation software.

ARGUMENTS:

 - A rthm-seq-palette object.

OPTIONAL ARGUMENTS:

 - <chop-points>. A list of integer pairs, each of which delineates a
   segment of the beat of the given rthm-seq-bar objects within the given
   rthm-seq-palette object, measured in the rhythmic unit specified by the
   <unit> argument. See the documentation for rthm-seq-bar::chop for more
   details.
 - <unit>. The rhythmic duration that serves as the unit of measurement for
   the chop points. Default = 's.
 - <number-bars-first>. T or NIL. This argument helps in naming (and
   therefore debugging) the newly-created bars. If T, the bars in the
   original rthm-seq will be renumbered, starting from 1, and this will be
   reflected in the tag of the new bars. E.g. if T, a new bar's tag may be
   new-bar-from-rs1-b3-time-range-1.750-to-2.000, if NIL this would be
   new-bar-from-rs1-time-range-1.750-to-2.000. Default = T.

RETURN VALUE:

 A rthm-seq-palette with the same top-level structure of the first argument,
 but each ID now referencing a sub-rthm-seq-palette with sequentially
 numbered rthm-seqs for each of the chopped results.

EXAMPLE:

```
;;; Create a rthm-seq-palette object, chop it with user-defined chop-points and
;;; a <unit> value of 'e, and print-simple the results
(let* ((rsp-orig (make-rsp
                  'sl-rsp
                  '((1
                     ((((2 4) (e) e (e) e))
                      :pitch-seq-palette (1 8)))
```

```
                           (2
                            ((((2 4) (s) e s e. (s)))
                             :pitch-seq-palette (3 5 7)))
                           (3
                            ((((3 4) q +s e. +q))
                             :pitch-seq-palette (1 7))))))
              (rsp-chopped (chop rsp-orig
                                 '((1 1) (1 2) (2 2))
                                 'e)))
        (print-simple rsp-chopped))

=>
rthm-seq-palette SL-RSP
rthm-seq-palette 1
rthm-seq 1
(1 8): rest 8,
rthm-seq 2
(1 4): rest E, NIL E,
rthm-seq 3
(1 8): NIL E,
rthm-seq 4
(1 8): rest 8,
rthm-seq 5
(1 4): rest E, NIL E,
rthm-seq 6
(1 8): NIL E,
rthm-seq 1
(1 8): rest S, NIL S,
rthm-seq 2
(1 4): rest S, NIL E, NIL S,
rthm-seq 3
(1 8): rest S, NIL S,
rthm-seq 4
(1 8): NIL E,
rthm-seq 5
(1 4): NIL E., rest S,
rthm-seq 6
(1 8): rest 8,
rthm-seq 1
(1 8): NIL E,
rthm-seq 2
(1 4): NIL Q,
rthm-seq 3
(1 8): rest 8,
rthm-seq 4
(1 8): rest S, NIL S,
```

```
rthm-seq 5
(1 4): rest S, NIL E.,
rthm-seq 6
(1 8): rest 8,
rthm-seq 7
(1 8): rest 8,
rthm-seq 8
(1 4): rest 4,
rthm-seq 9
(1 8): rest 8,
```

SYNOPSIS:

```
(defmethod chop ((rsp rthm-seq-palette) &optional chop-points
                                         (unit 's)
                                         (number-bars-first t))
```

28.5.300   rthm-seq-palette/cmn-display

[rthm-seq-palette] [Methods]

DESCRIPTION:

 Generate printable music notation output (.EPS) of the given
 rthm-seq-palette object using the Common Music Notation (CMN)
 interface. The method requires at least the name of the given
 rthm-seq-palette object to set, but has several additional optional
 arguments for customizing output.

 NB: Most of the keyword arguments are CMN attributes and share the same
     name as the CMN feature they effect.

ARGUMENTS:

 - A rthm-seq-palette object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :all-output-in-one-file. T or NIL to indicate whether to write the output
   to a multi-page file or to separate files for each page.
   T = one multi-page file. Default = T. This is a direct CMN attribute.
 - :file. The file path, including the file name, of the file to be
   generated.
 - :staff-separation. A number to indicate the amount of white space between

staves belong to the same system, measured as a factor of the staff
    height. Default = 3. This is a direct CMN attribute.
  - :line-separation. A number to indicate the amount of white space between
    lines of music (systems), measured as a factor of the staff
    height. Default = 5. This is a direct CMN attribute.
  - :page-nums. T or NIL to indicate whether or not to print page numbers on
    the pages. T = print page numbers. Default = T.
  - :no-accidentals. T or NIL to indicate whether or not to supress printing
    accidentals for each and every note (rather than once per bar).
    T = supress printing all accidentals. Default = NIL.
  - :seqs-per-system. An integer indicating the number of rthm-seq objects to
    be printed in one staff system. Default = 1.
  - :size. A number to indicate the font size of the CMN output.
  - :auto-open. Automatically open the EPS file?.
    Default = (get-sc-config cmn-display-auto-open)

RETURN VALUE:

 T

EXAMPLE:

```
;; A typical example with some specified keyword values for file and size
(let ((mrsp
       (make-rsp 'rsp-test
                 '((seq1 ((((2 4) q +e. s)
                           ((s) e (s) q)
                           (+e. s { 3 (te) te te } ))
                          :pitch-seq-palette ((1 2 3 4 5 6 7)
                                              (1 3 5 7 2 4 6)
                                              (1 4 2 6 3 7 5)
                                              (1 5 2 7 3 2 4))))
                   (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                           ({ 3 +te (te) te } e e (h)))
                          :pitch-seq-palette (2 3 4 5 6 7 8)))
                   (seq3 ((((2 4) e e { 3 te te te })
                           ((4 4) (e) e e e s s (s) s q))
                          :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
  (cmn-display mrsp
               :file "/tmp/rmsp-output.eps"
               :size 10))
```

SYNOPSIS:

```
(defmethod cmn-display ((rsp rthm-seq-palette)
```

```
                                &key
                                (all-output-in-one-file t)
                                (file
                                 (format nil "~a~a.eps"
                                         (get-sc-config 'default-dir)
                                         (string-downcase (id rsp))))
                                (staff-separation 3)
                                (line-separation 5)
                                (page-nums t)
                                (no-accidentals t)
                                (seqs-per-system 1)
                                (auto-open (get-sc-config 'cmn-display-auto-open))
                                (size 15))
```

28.5.301   rthm-seq-palette/create-psps

[rthm-seq-palette] [Methods]

DATE:

 30 Mar 2006

DESCRIPTION:

 Automatically create pitch-seq-palette objects for each rthm-seq object in
 the given rthm-seq-palette object.

 The selection function given as an optional keyword argument should be able
 to generate a list of numbers (relative note levels) for a rthm-seq of any
 length; it takes two arguments only: the number of notes needed and the
 pitch-seq data lists (see below).

 As a pitch-seq-palette usually has several options for each rthm-seq
 object, it's best when the selection-fun doesn't always return the same
 thing given the same number of notes. NB: This will silently kill the data
 of any pitch-seq-palette objects already supplied for any rthm-seqs in the
 palette.

 Note that the default selection function will suffice in lots of cases.
 However, you may just want to use different data lists with the default
 function. In that case just pass these via :selection-fun-data.

ARGUMENTS:

 - A rthm-seq-palette object.

OPTIONAL ARGUMENTS:

```
keyword arguments
- :selection-fun. This is a function that will return the pitch-seq
  numbers. It takes two arguments only: 1) the number of notes needed, and
  2) the pitch-seq data lists. The function also needs to be able to handle
  being passed NIL NIL as arguments. In this case it should reset, if needs
  be; i.e. it's just a call to init and should return nothing. Default =
  #'create-psps-default.
- :pitch-seqs-per-rthm-seq. This is an integer that is the number of
  pitch-seqs each rthm-seq should have. NB: The method will simply cycle
  through the pitch-seqs given in the selection function to create the
  required number.  Default = 3.
- :selection-fun-data. This contains the pitch-seq lists to be passed to
  the default selection function. There can be as many pitch-seqs in these
  lists as desired. The number of notes the pitch-seq will provide is the
  first item of the list. They need not be in ascending order. When this
  argument is passed a value of T, the selection function will reinitialize
  its default data and use that.
- :overwrite. If NIL and a pitch-seq-palette was given for a particular
  rthm-seq, then a new one won't be created. Default = T.
- :reinit. Used internally.  Do not change.

  At the moment, the default data are:
          '((1 ((3) (3) (1) (25)))
            (2 ((3 4) (5 2) (25 25) (1 25)))
            (3 ((3 4 3) (5 9 6) (1 2 4) (5 2 2) (6 2 3)))
            (4 ((3 4 3 4) (5 3 6 4) (9 4 5 11) (2 10 4 8)))
            (5 ((5 5 6 5 8) (7 7 7 4 8) (11 8 4 10 2) (7 7 4 9 9)))
            (6 ((4 5 5 3 6 6) (3 8 3 9 3 8) (9 3 9 5 10 6)))
            (7 ((8 8 8 5 9 6 9 ) (9 3 8 4 7 5 4) (3 4 3 5 3 4 3)))
            (8 ((3 3 4 3 3 1 5 4) (10 3 9 3 8 3 7 4) (3 5 8 2 8 9 4 11)))
            (9 ((3 6 4 7 4 7 3 6 7) (10 2 9 2 8 2 7 2 3)
                (2 9 3 9 4 9 9 6 11)))
            (10 ((9 9 9 3 9 9 3 5 9 5) (8 9 8 9 5 9 9 5 6 6)))
            (12 ((1 2 5 5 5 5 5 5 5 4 5) (2 1 5 1 5 1 6 5 1 5 2 5)))
            (13 ((1 2 5 5 5 5 5 5 5 4 5 2) (2 1 5 1 5 1 6 5 1 5 2 5 1)))
            (14 ((1 2 5 5 5 5 5 5 5 4 5 2 1)
                 (2 1 5 1 5 1 6 5 1 5 2 5 1 2)))
            (15 ((1 2 5 5 5 5 5 5 5 4 5 2 1 2)
                 (2 1 5 1 5 1 6 5 1 5 2 5 1 2 6))))))
```

RETURN VALUE:

```
Always returns the rthm-seq-palette argument.
```

EXAMPLE:

```
;; Create a rthm-seq-palette object that specifies pitch-seq-palettes for each
;; contained rthm-seq object and print the values of the individual
;; pitch-seq-palettes. Then apply the create-psps method using its default
;; values, and print the values of the individual pitch-seq-palettes again to
;; see the change.  NB You wouldn't normally specify pitch-seq-palettes in your
;; rthm-seq-palette as the whole point of this method is to have them created
;; algorithmically, but they are given here for purposes of comparison.
(let ((mrsp (make-rsp 'rsp-test
                      '((seq1 ((((2 4) q +e. s)
                                ((s) e (s) q)
                                (+e. s { 3 (te) te te } ))
                               :pitch-seq-palette (1 2 3 4 5 6 7)))
                        (seq2 ((((3 4) (e.) s { 3 te te te } +q)
                                ({ 3 +te (te) te } e e (q)))
                               :pitch-seq-palette (2 3 4 5 6 7 8)))
                        (seq3 ((((2 4) e e { 3 te te te })
                                ((5 8) (e) e e e s s))
                               :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2)))))))
  (print
   (loop for rs in (data mrsp)
      collect
        (loop for ps in (data (pitch-seq-palette rs))
           collect (data ps))))
  (create-psps mrsp)
  (print
   (loop for rs in (data mrsp)
      collect
        (loop for ps in (data (pitch-seq-palette rs))
                collect (data ps)))))

=>
(((1 2 3 4 5 6 7)) ((2 3 4 5 6 7 8)) ((3 4 5 6 7 8 9 10 1 2)))

(((8 8 8 5 9 6 9) (9 3 8 4 7 5 4) (3 4 3 5 3 4 3))
 ((8 8 8 5 9 6 9) (9 3 8 4 7 5 4) (3 4 3 5 3 4 3))
 ((9 9 9 3 9 9 3 5 9 5) (8 9 8 9 5 9 9 5 6 6) (9 9 9 3 9 9 3 5 9 5)))

;; Use the :pitch-seqs-per-rthm-seq keyword argument to specify the number of
;; pitch-seq objects to be created for each rthm-seq. This example creates 5
;; instead of the default 3.
(let ((mrsp (make-rsp 'rsp-test
                      '((seq1 ((((2 4) q +e. s)
                                ((s) e (s) q)
                                (+e. s { 3 (te) te te } ))))
```

```
                              (seq2 ((((3 4) (e.) s { 3 te te te } +q)
                                     ({ 3 +te (te) te } e e (q)))))
                              (seq3 ((((2 4) e e { 3 te te te })
                                     ((5 8) (e) e e e s s))))))))))
  (create-psps mrsp :pitch-seqs-per-rthm-seq 5)
  (loop for rs in (data mrsp)
            collect
              (loop for ps in (data (pitch-seq-palette rs))
                 collect (data ps))))

=>
(((8 8 8 5 9 6 9) (9 3 8 4 7 5 4) (3 4 3 5 3 4 3) (8 8 8 5 9 6 9)
  (9 3 8 4 7 5 4))
 ((3 4 3 5 3 4 3) (8 8 8 5 9 6 9) (9 3 8 4 7 5 4) (3 4 3 5 3 4 3)
  (8 8 8 5 9 6 9))
 ((9 9 9 3 9 9 3 5 9 5) (8 9 8 9 5 9 9 5 6 6) (9 9 9 3 9 9 3 5 9 5)
  (8 9 8 9 5 9 9 5 6 6) (9 9 9 3 9 9 3 5 9 5)))

;;; Now an example with our own selection-fun creating random pitch-seqs for
;;; demo purposes only:
(let ((mrsp (make-rsp 'rsp-test
                      '((seq1 ((((2 4) q +e. s)
                                ((s) e (s) q)
                                (+e. s { 3 (te) te te }))))
                        (seq2 ((((3 4) (e.) s { 3 te te te } +q)
                                ({ 3 +te (te) te } e e (q)))))
                        (seq3 ((((2 4) e e { 3 te te te })
                                ((5 8) (e) e e e s s))))))))
  (create-psps
   mrsp
   :selection-fun #'(lambda
                        (num-notes data-lists)
                      ;; NB we're not doing anything with data-lists here
                      (loop repeat num-notes collect (random 10))))
   (loop for rs in (data mrsp)
      collect
        (loop for ps in (data (pitch-seq-palette rs))
                collect (data ps))))
=>
(((5 4 3 0 3 8 0) (1 2 8 5 3 7 8) (5 5 7 3 9 1 7))
 ((3 1 5 6 6 7 1) (7 7 8 5 5 2 4) (9 1 3 0 8 7 8))
 ((4 8 6 9 6 6 0 8 1 2) (1 5 5 7 7 2 9 3 1 2) (1 5 6 2 5 3 7 3 4 2)))
```

SYNOPSIS:

```
(defmethod create-psps ((rsp rthm-seq-palette)
```

```
                               &key
                                 (selection-fun #'create-psps-default)
                                 (selection-fun-data nil)
                                 ;; MDE Sat Jul 14 17:20:27 2012 --
                                 (reinit t)
                                 ;; MDE Thu Feb  1 17:34:42 2018
                                 (overwrite t)
                                 ;; MDE Mon Jun 22 15:31:48 2015
                                 (chords t)
                                 (pitch-seqs-per-rthm-seq 3))
```

28.5.302   rthm-seq-palette/create-psps-default

[rthm-seq-palette] [Functions]

ARGUMENTS:

- An integer that is the number of notes for which a pitch-seq-palette
  object is needed.
- the pitch-seq data (see documentation for create psps method). Ideally
  this would only be passed the first time the function is called.

RETURN VALUE:

 A list of numbers suitable for use in creating a pitch-seq object.

SYNOPSIS:

```
  (defun create-psps-default (num-notes data-lists)
```

28.5.303   rthm-seq-palette/get-multipliers

[rthm-seq-palette] [Methods]

DESCRIPTION:

 Get a list of factors by which a specified rhythmic unit must be multiplied
 in order to create the rhythms of a specified rthm-seq object within the
 given rthm-seq-palette object.

 See also rthm-seq method for more information.

ARGUMENTS:

 - A rthm-seq object.

- A rhythm unit, either as a number of a CMN shorthand symbol (i.e. 'e)
- A symbol that is the ID of the rthm-seq-object for which the multipliers
  is sought is also a required argument (though it is listed as an optional
  argument for internal reasons).

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to round the results. T = round.
  Default = NIL. NB: Lisp always rounds to even numbers, meaning x.5 may
  sometimes round up and sometimes round down; thus (round 1.5) => 2, and
  (round 2.5) => 2.

RETURN VALUE:

A list of numbers.

EXAMPLE:

```
;; Returns a list of numbers, by default not rounded
(let ((mrsp
       (make-rsp 'rsp-test
                 '((seq1 ((((2 4) q +e. s)
                           ((s) e (s) q)
                           (+e. s { 3 (te) te te } ))
                          :pitch-seq-palette ((1 2 3 4 5 6 7)
                                              (1 3 5 7 2 4 6)
                                              (1 4 2 6 3 7 5)
                                              (1 5 2 7 3 2 4))))
                   (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                           ({ 3 +te (te) te } e e (h)))
                          :pitch-seq-palette (2 3 4 5 6 7 8)))
                   (seq3 ((((2 4) e e { 3 te te te })
                           ((4 4) (e) e e e s s (s) s q))
                          :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
  (get-multipliers mrsp 'e 'seq1))

=> (2.0 1.0 1.5 2.0 1.1666666666666665 0.6666666666666666 0.6666666666666666)

;; Setting the option <round> argument to T returns rounded results
(let ((mrsp
       (make-rsp 'rsp-test
                 '((seq1 ((((2 4) q +e. s)
                           ((s) e (s) q)
                           (+e. s { 3 (te) te te } ))
                          :pitch-seq-palette ((1 2 3 4 5 6 7)
```

```
                                                  (1 3 5 7 2 4 6)
                                                  (1 4 2 6 3 7 5)
                                                  (1 5 2 7 3 2 4))))
                  (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                         ({ 3 +te (te) te } e e (h)))
                        :pitch-seq-palette (2 3 4 5 6 7 8)))
                  (seq3 ((((2 4) e e { 3 te te te })
                         ((4 4) (e) e e e s s (s) s q))
                        :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
  (get-multipliers mrsp 'e 'seq1 t))

=> (2 1 2 2 1 1 1)

;; The ID argument is required, even though it's listed as being optional. The
;; method interrupts with an error if no ID is supplied
(let ((mrsp
        (make-rsp 'rsp-test
                  '((seq1 ((((2 4) q +e. s)
                           ((s) e (s) q)
                           (+e. s { 3 (te) te te } ))
                          :pitch-seq-palette ((1 2 3 4 5 6 7)
                                              (1 3 5 7 2 4 6)
                                              (1 4 2 6 3 7 5)
                                              (1 5 2 7 3 2 4))))
                  (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                         ({ 3 +te (te) te } e e (h)))
                        :pitch-seq-palette (2 3 4 5 6 7 8)))
                  (seq3 ((((2 4) e e { 3 te te te })
                         ((4 4) (e) e e e s s (s) s q))
                        :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
  (get-multipliers mrsp 'e))

=>
rthm-seq-palette::get-multipliers: third argument (rthm-seq ID) is required.
   [Condition of type SIMPLE-ERROR]

;;; Applying the method to the a multiple-bar rthm-seq object may return
;;; different results than applying the method to each of the bars contained
;;; within that rthm-seq object as individual one-bar rthm-seq objects, as the
;;; method measures the distances between attacked notes, regardless of ties
;;; and rests.
(let ((rs1 (make-rthm-seq '(seq1 ((((2 4) q +e. s))
                                   :pitch-seq-palette ((1 2))))))
      (rs2 (make-rthm-seq '(seq2 ((((2 4) (s) e (s) q))
                                   :pitch-seq-palette ((1 2))))))
      (rs3 (make-rthm-seq '(seq3 ((((2 4) +e. s { 3 (te) te te } ))
```

```
                                       :pitch-seq-palette ((1 2 3))))))
      (rs4 (make-rthm-seq '(seq4 ((((2 4) q +e. s)
                                   ((s) e (s) q)
                                   (+e. s { 3 (te) te te } ))
                                  :pitch-seq-palette ((1 2 3 4 5 6 7)))))))))
  (print (get-multipliers rs1 'e))
  (print (get-multipliers rs2 'e))
  (print (get-multipliers rs3 'e))
  (print (get-multipliers rs4 'e)))

=>
(3.5 0.5)
(1.5 2.0)
(1.1666666666666665 0.6666666666666666 0.6666666666666666)
(3.5 1.0 1.5 3.5 1.1666666666666665 0.6666666666666666 0.6666666666666666)
```

SYNOPSIS:

```
(defmethod get-multipliers ((rsp rthm-seq-palette) rthm &optional id round)
```

28.5.304   rthm-seq-palette/make-rsp

[rthm-seq-palette] [Functions]

DESCRIPTION:

 Create a rthm-seq-palette object.

ARGUMENTS:

 - A symbol that is to be the ID of the rhtm-seq-palette object created.
 - A list containing rhtm-seq data to be made into rthm-seqs. Each item in
   this list is a list of data formatted as it would be when passed to the
   make-rthm-seq function.

OPTIONAL ARGUMENTS:

 T or NIL to indicate whether to automatically generate and store inversions
 of the pitch-seq-palette passed to the rthm-seq objects in the
 rthm-seq-palette object created. T = generate and store. Default = NIL.

RETURN VALUE:

 A rthm-seq-palette object.

EXAMPLE:

```
(make-rsp 'rsp-test '((seq1 ((((2 4) q +e. s)
                             ((s) e (s) q)
                             (+e. s { 3 (te) te te } ))
                            :pitch-seq-palette (1 7 3 4 5 2 6)))
                      (seq2 ((((3 4) (e.) s { 3 te te te } +q)
                             ({ 3 +te (te) te } e e (q)))
                            :pitch-seq-palette (3 1 2 5 1 7 6)))
                      (seq3 ((((2 4) e e { 3 te te te })
                             ((5 8) (e) e e e s s))
                            :pitch-seq-palette (4 4 4 5 4 4 4 5 4 3)))))

=>
RTHM-SEQ-PALETTE: psp-inversions: NIL
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 3
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: RSP-TEST, tag: NIL,
data: (
RTHM-SEQ: num-bars: 3
[...]

;; Create two rthm-seq-palette objects, one with :psp-inversions set to NIL and
;; one with it set to T, and print the DATA of the pitch-seq-palettes of each
(let ((mrsp1 (make-rsp 'rsp-test
                       '((seq1 ((((2 4) q +e. s)
                                ((s) e (s) q)
                                (+e. s { 3 (te) te te } ))
                               :pitch-seq-palette (1 7 3 4 5 2 6))))
                       :psp-inversions nil))
      (mrsp2 (make-rsp 'rsp-test
                       '((seq1 ((((2 4) q +e. s)
                                ((s) e (s) q)
                                (+e. s { 3 (te) te te } ))
                               :pitch-seq-palette (1 7 3 4 5 2 6))))
                       :psp-inversions t)))
  (print
   (loop for i in (data (pitch-seq-palette (first (data mrsp1))))
      collect (data i)))
```

```
  (print
   (loop for i in (data (pitch-seq-palette (first (data mrsp2))))
      collect (data i))))
```

```
=>
((1 7 3 4 5 2 6))
((1 7 3 4 5 2 6) (7 1 5 4 3 6 2))
```

SYNOPSIS:

```
(defun make-rsp (id data &key (psp-inversions nil))
```

28.5.305   rthm-seq-palette/make-rthm-seq-from-unit-multipliers-simp

[rthm-seq-palette] [Functions]

DESCRIPTION:

```
 Make a rthm-seq-palette using lists of lists of multiples of a single unit
 e.g. '(2 2 1) would result in an automatic time signature of 5/8 if passed a
 unit of 'e (or 8). In that case the rhythms would be q q e. Rests may be
 created by passing a number in parentheses.
```

ARGUMENTS:

```
 - the ID for the rthm-seq-plaette
 - the rhythm unit (symbol, number or rhythm object)
 - the lists of lists multipliers, each sublist representing a rthm-seq and
   each sublist of that representing a bar.
```

RETURN VALUE:

```
 a rthm-seq-palette object
```

EXAMPLE:

```
(print-simple (make-rsp-from-unit-multipliers-simp
              'test-unit-mults 's
              '(((3 4 1 2) (2 (1) 2))
                ((2 1 2 1) (3) ((4)))
                (((3) 4 1) (3 4 1) ((2) 3) (2 3) (2)))))
rthm-seq-palette TEST-UNIT-MULTS
rthm-seq 1
NIL: bar -1: (10 16):
note E.,
```

```
note Q,
note S,
note E,
NIL: bar -1: (5 16):
note E,
rest S,
note E,
rthm-seq 2
NIL: bar -1: (6 16):
note E,
note S,
note E,
note S,
NIL: bar -1: (3 16):
note E.,
NIL: bar -1: (4 16):
rest 4,
rthm-seq 3
NIL: bar -1: (8 16):
rest E.,
note Q,
note S,
NIL: bar -1: (8 16):
note E.,
note Q,
note S,
NIL: bar -1: (5 16):
rest E,
note E.,
NIL: bar -1: (5 16):
note E,
note E.,
NIL: bar -1: (2 16):
note E,
T
```

SYNOPSIS:

```
(defun make-rsp-from-unit-multipliers-simp (id unit seqs)
```

28.5.306   rthm-seq-palette/reset-psps

[rthm-seq-palette] [Methods]

DESCRIPTION:

Call the reset method (inherited from circular-sclist) for all
pitch-seq-palette objects of all rthm-seq objects in the given
rthm-seq-palette object, resetting their pointers to the head of the
sequence. This ensures that each rthm-seq starts over again at the first
note of the first given pitch-seq.

NB Since 28/10/13 there's also a reset method for the palette class which
will mean we call the reset method of all pitch-seqs here. So this method
is now partially obsolete (but still perhaps useful in that in only resets
the psps, not the whole palette).

ARGUMENTS:

 - A rthm-seq-palette object.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;; Create a rthm-seq-palette object whose first rthm-seq has three pitch-seq
;; objects in its pitch-seq-palette. Apply the get-next method to the
;; pitch-seq-palette object of the first rthm-seq object twice, then print the
;; data of the next pitch-seq object to show where we are. Apply the reset-psps
;; method and print the data of the next pitch-seq object to show that we've
;; returned to the beginning of the pitch-seq-palette.

(let ((mrsp
       (make-rsp 'rsp-test
                   '((seq1 ((((2 4) q +e. s)
                            ((s) e (s) q)
                            (+e. s { 3 (te) te te } ))
                           :pitch-seq-palette ((1 2 3 4 5 6 7)
                                               (1 3 5 7 2 4 6)
                                               (1 4 2 6 3 7 5)
                                               (1 5 2 7 3 2 4))))
                    (seq2 ((((3 4) (e.) s { 3 te te te } +q)
                           ({ 3 +te (te) te } e e (q)))
                           :pitch-seq-palette (2 3 4 5 6 7 8)))
                    (seq3 ((((2 4) e e { 3 te te te })
                           ((5 8) (e) e e e s s))
                           :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2)))))))
  (loop repeat 2
       do (get-next (pitch-seq-palette (first (data mrsp)))))
```

```
  (print (data (get-next (pitch-seq-palette (first (data mrsp))))))
  (reset-psps mrsp)
  (print (data (get-next (pitch-seq-palette (first (data mrsp)))))))))
```

```
=>
(1 4 2 6 3 7 5)
(1 2 3 4 5 6 7)
```

SYNOPSIS:

```
(defmethod reset-psps ((rsp rthm-seq-palette))
```

28.5.307   rthm-seq-palette/rs-equal-durations?

[rthm-seq-palette] [Methods]

DATE:

 June 13th 2017, Edinburgh

DESCRIPTION:

 See if all the rthm-seqs in a palette (or sub-palette) have the same
 duration.

ARGUMENTS:

 - the rthm-seq-palette object

OPTIONAL ARGUMENTS:

 - a reference (symbol or list) into the rthm-seq-palette to examine a
   sub-palette

RETURN VALUE:

 If all rthm-seqs in the palette have the same duration, then their duration
 in quarter notes, otherwise nil.

SYNOPSIS:

```
(defmethod rs-equal-durations? ((rsp rthm-seq-palette) &optional ref)
```

28.5.308   rthm-seq-palette/rsp-subseq

[rthm-seq-palette] [Methods]

DATE:

 23rd October 2013

DESCRIPTION:

 (Recursively) change all the rthm-seq objects in the palette to be a
 subsequence of the existing rthm-seqs.

ARGUMENTS:

 - the original rthm-seq-palette object
 - the start bar (1-based)

OPTIONAL ARGUMENTS:

 - the end bar (1-based and (unlike Lisp's subseq function) inclusive). If
   NIL, we'll use the original end bar of each rthm-seq. Default = NIL.

RETURN VALUE:

 The original rthm-seq-palette but with the new rthm-seqs

SYNOPSIS:

(defmethod rsp-subseq ((rsp rthm-seq-palette) start &optional end)

28.5.309   rthm-seq-palette/scale

[rthm-seq-palette] [Methods]

DESCRIPTION:

 Scale the durations of the rhythm objects in a given rthm-seq-palette
 object by the specified factor.

 NB: As is evident in the examples below, this method does not replace the
     original data in the rthm-seq-palette object's DATA slot.

ARGUMENTS:

- A rthm-seq-palette object.
- A real number that is the scaling factor.

OPTIONAL ARGUMENTS:

 (- the three IGNORE arguments are for internal purposes only).

RETURN VALUE:

 Returns a rthm-seq-palette object.

EXAMPLE:

```
;; Returns a rthm-seq-palette object
(let ((mrsp
       (make-rsp 'rsp-test
                 '((seq1 ((((2 4) q +e. s)
                           ((s) e (s) q)
                           (+e. s { 3 (te) te te } ))
                          :pitch-seq-palette ((1 2 3 4 5 6 7)
                                              (1 3 5 7 2 4 6)
                                              (1 4 2 6 3 7 5)
                                              (1 5 2 7 3 2 4))))
                   (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                           ({ 3 +te (te) te } e e (h)))
                          :pitch-seq-palette (2 3 4 5 6 7 8)))
                   (seq3 ((((2 4) e e { 3 te te te })
                           ((4 4) (e) e e e s s (s) s q))
                          :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
   (scale mrsp 2))

=>
RTHM-SEQ-PALETTE: psp-inversions: NIL
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 3
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: RSP-TEST, tag: NIL,
data: (
RTHM-SEQ: num-bars: 3
```

```
;; Apply the method and loop through the rthm-seq objects in the
;; rthm-seq-palette object's DATA slot, using the print-simple method to see
;; the changes

(let ((mrsp
       (make-rsp 'rsp-test
                  '((seq1 ((((2 4) q +e. s)
                            ((s) e (s) q)
                            (+e. s { 3 (te) te te } ))
                           :pitch-seq-palette ((1 2 3 4 5 6 7)
                                               (1 3 5 7 2 4 6)
                                               (1 4 2 6 3 7 5)
                                               (1 5 2 7 3 2 4))))
                    (seq2 ((((4 4) (e.) s { 3 te te te } +h)
                            ({ 3 +te (te) te } e e (h)))
                           :pitch-seq-palette (2 3 4 5 6 7 8)))
                    (seq3 ((((2 4) e e { 3 te te te })
                            ((4 4) (e) e e e s s (s) s q))
                           :pitch-seq-palette (3 4 5 6 7 8 9 10 1 2 3 7)))))))
   (scale mrsp .5)
   (print-simple mrsp))

=>
rthm-seq-palette RSP-TEST
rthm-seq SEQ1
(2 8): note E, note S., note 32,
(2 8): rest 32, note S, rest 32, note E,
(2 8): note S., note 32, rest TS, note TS, note TS,
rthm-seq SEQ2
(4 8): rest S., note 32, note TS, note TS, note TS, note Q,
(4 8): note TS, rest TS, note TS, note S, note S, rest Q,
rthm-seq SEQ3
(2 8): note S, note S, note TS, note TS, note TS,
(4 8): rest S, note S, note S, note S, note 32, note 32, rest 32, note 32, note E,
```

SYNOPSIS:

```
(defmethod scale ((rsp rthm-seq-palette) scaler
                  &optional ignore1 ignore2 ignore3)
```

28.5.310   rthm-seq-palette/split-into-single-bars

[rthm-seq-palette] [Methods]

DESCRIPTION:

Split every rthm-seq in a palette into as many single-bar rthm-seqs as
there are bars. This creates an extra level of recursion so that whereas a
rthm-seq that was referenced by e.g. '(long 1 a) beforehand, afterwards
each bar will be reference by '(long 1 a 1), '(long 1 a 2) etc. This is
true even for sequences that only contained one bar before processing. The
pitch-seq-palettes of the original rthm-seqs will be used to set the
pitch-seqs of the new rthm-seqs.

ARGUMENTS:

- the rthm-seq-palette object

OPTIONAL ARGUMENTS:

- whether to clone the palette before processing (T or NIL). Default = T.

RETURN VALUE:

A rthm-seq-palette object with extra layers of recursion.

SYNOPSIS:

(defmethod split-into-single-bars ((rsp rthm-seq-palette) &optional (clone t))

28.5.311   palette/set-palette

[palette] [Classes]

NAME:

set-palette

File:	set-palette.lsp
Class Hierarchy:	named-object -> linked-named-object -> sclist -> circular-sclist -> assoc-list -> recursive-assoc-list -> palette -> set-palette
Version:	1.0.11
Project:	slippery chicken (algorithmic composition)
Purpose:	Implementation of the set-palette class which extends the palette class by simply instantiating the sets given in the palette.

Note that the sets in this palette may refer to
previously defined sets in order to obviate retyping note
lists. Hence the reference to bcl-chord2 in the
bcl-chord3 set of the example below will instantiate a
set based on a transposed clone of that set previously
stored as bcl-chord2.

```
(make-set-palette
 'test
 '((bcl-chord1
    ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5
          dqf5 gs5 b5)
     :subsets
     ((tc1 (ds2 e3 a4))
      (tc2 (bf1 d4 cqs5))
      (qc1 (aqf2 e3 a4 dqf5 b5))
      (qc2 (bf1 c3 gqs3 cs4 cqs5)))
     :related-sets
     ((missing (bqs0 eqs1 f5 aqs5 eqf6 fqs6
                     bqf6 dqs7 fs7)))))
   (bcl-chord2
    ((bf1 d2 fqf2 fqs2 b2 c3 f3 g3 bqf3 bqs3 fs4 gs4 a4
      cs5 gqf5)
     :subsets
     ((tc1 (d2 g3 cs5))
      (tc2 (eqs2 f3 bqf3))
      (qc1 (eqs2 c3 f3 fs4 gqf5))
      (qc2 (d2 fqs2 bqs3 gs4 a4)))
     :related-sets
     ((missing (aqs0 dqs1 ds5 gqs5 dqf6 eqf6 aqf6 cqs7
                     e7)))))
   (bcl-chord3
    (bcl-chord2 :transposition 13))))
```

Author: Michael Edwards: m@michael-edwards.org

Creation date: August 14th 2001

$$ Last modified: 17:06:17 Fri Jun 5 2020 CEST

SVN ID: Id

28.5.312   set-palette/add-harmonics

[set-palette] [Methods]

DESCRIPTION:

 Add harmonically-related pitches to each set in the palette. See the sc-set
 class method for details and the get-harmonics function (utilities.lsp) for
 keyword arguments.

SYNOPSIS:

```
(defmethod add-harmonics ((sp set-palette) &rest keywords)
```

28.5.313   set-palette/auto-sequence

[set-palette] [Methods]

DATE:

 July 30th 2015, Edinburgh

DESCRIPTION:

 Automatically create an ordering for the sets in a set-palette based on a
 dissonance envelope and a spectral centroid envelope. For a description of
 a set or chord's dissonance or spectral centroid see the
 calculate-spectral-centroid and calculate-dissonance methods in the chord
 class.

 The envelopes describe a desired general tendency to, for example, proceed
 from less dissonant to more dissonant sets as the ordering (sequence) of
 sets proceeds. Such an envelope would move from lower to higher
 values. Similarly with the spectral centroid envelope: moving from a lower
 to a higher value implies moving from chords with an overall lower pitch
 height to chords with a higher pitch height.

 The envelopes should be expressed over any X axis range but with a Y axis
 range of 0.0 to 1.0 only. The x-axes will be stretched to fit over the
 number of sets in the palette. The Y axes will be stretched to fit the
 range of dissonance and centroid values to be found in the sets in the
 palette (see the quality-extremes method).

 The default method is successive. It selects sets one-by-one via a sort
 function which compares the sets' characteristics to those of the current
 envelope values. Essentially, when we are comparing two sets via the sort

function, we look at the deviation between the sets' dissonance values and
the current desired dissonance value from the envelope, similarly with the
spectral centroid. The chord with the smallest combined deviation will be
chosen first. If either envelope is nil, then the sorting is based on the
other envelope only. Similarly, weighting factors of any arbitrary positive
number can be passed via :dissonance-weight and :centroid-weight to
emphasise or deemphasise these properties when sorting. Higher values will
mean that that property will take precedence over the other property.

An alternative method, indicated by :permutate T, is to get all (or at
least a lot of) permutations of the sets in the palette and then score each
ordering against the curves, as a form of fitness test. The advantage of
this approach is that the best solution overall can be found. The
successive approach, on the other hand will find the best fit at the
beginning of the process but as it proceeds along the envelope and we run
out of sets to select from, the fit can become worse and worse.

In any case there is no guarantee that the desired curves will be expressed
exactly in the returned ordering. The function tries to find the best fit
but success depends very much on the number and variety of sets in the
palette.

Also taken into account when in both methods is the lowest note of the
set. If :repeating-bass is NIL (the default) then the function tries to
avoid repeating bass notes in two consecutive sets. Repeating highest notes
are allowed.

NB Unlike the chord methods which calculate dissonance and spectral
centroid, there is no way to pass spectral data here. If you want to
override the default spectral data then use (set-sc-config 'default-spectra
...). See globals.lsp for more details.

ARGUMENTS:

- the set-palette object

OPTIONAL ARGUMENTS:

keyword arguments:
- dissonance-env: The desired dissonance envelope. Y values: 0.0 to 1.0
  Default '(0 .1 62 1 100 .3)
- centroid-env: The desired spectral centroid envelope.Y values: 0.0 to 1.0
  Default '(0 .4 62 1 100 .2)
- dissonance-weight: A weighting factor (scaler) applied to
  dissonance. Higher values result in dissonance playing a larger role in
  the decision process. Default 1.0

- centroid-weight: a similar factor for spectral centroid. Default 1.0
- verbose: Whether to print data as the decision process proceeds.
  Default NIL
- repeating-bass: Whether to allow bass notes to repeat between two
  consecutive chords. Default NIL
- silent: Whether to print warnings or not. Default NIL
- map-section: if you want a list of references suitable to be passed
  as a set-map, set this keyword to the integer ID of the section the map
  will be used for. Default = NIL.
- permutate: use the permutation rather than the successive approach. This
  can either be T (which will limit to 2000), 'all (get all
  permutations--could take a very long time), or an integer to represent
  the maximum number of permutations we'll try. Default = NIL.

RETURN VALUE:

If we're using the permutate method then a single list of the sets'
set-palette references is returned. If we're using the successive method we
return two values: A list of the full references of the sets in the set
palette in the automatically determined order, along with a list of the
deviations from the ideal this order represents. Either method might return
a list suitable for passing to a set-map (see example below).

EXAMPLE:

```
(auto-sequence
 (recursive-set-palette-from-ring-mod '(a4 b4) 'spfrm-test
                                      :warn-no-bass nil)
 :verbose nil :centroid-weight 2 :silent t)
=>
((A4 15) (A4 13) (A4 4) (A4 7) (B4 4) (A4 14) (A4 2) (A4 20) (B4 2) (B4 17)
 (B4 14) (B4 11) (B4 9) (A4 12) (B4 21) (B4 6) (A4 6) (B4 5) (B4 7) (B4 12)
 (A4 19) (A4 5) (A4 9) (A4 3) (A4 11) (A4 10) (B4 18) (B4 10) (B4 16) (B4 3)
 (B4 19) (A4 17) (B4 13) (A4 8) (B4 15) (B4 8) (B4 20) (B4 1) (A4 21) (A4 1)
 (A4 18) (A4 16))
((0.3433064594719475d0 0.04460029910249326d0 0.38790675857444074d0)
 (0.36511199730074173d0 0.10718227303159739d0 0.4722942703323391d0)
...

(auto-sequence
 (recursive-set-palette-from-ring-mod '(a4 b4) 'spfrm-test
                                      :warn-no-bass nil)
 :verbose nil :centroid-weight 2 :silent t :permutate t :map-section 1)
=>
((1
  ((B4 1) (B4 3) (B4 15) (B4 5) (B4 10) (A4 3) (A4 17) (A4 1) (A4 14) (A4 15)
```

```
   (B4 12) (A4 4) (B4 2) (B4 18) (A4 20) (A4 5) (B4 4) (B4 6) (B4 20) (B4 17)
   (B4 16) (B4 11) (A4 6) (B4 9) (A4 7) (A4 12) (B4 21) (A4 18) (A4 19) (A4 13)
   (B4 19) (B4 7) (A4 16) (B4 13) (A4 8) (A4 2) (B4 8) (A4 9) (A4 10) (A4 11)
   (A4 21) (B4 14))))
```

SYNOPSIS:

```
(defmethod auto-sequence ((sp set-palette)
                          &key
                            (dissonance-env '(0 .1 62 1 100 .3))
                            (centroid-env '(0 .4 62 1 100 .2))
                            (dissonance-weight 1.0)
                            (centroid-weight 1.0)
                            map-section permutate verbose repeating-bass silent)
```

28.5.314   set-palette/calculate-spectral-centroid

[set-palette] [Methods]

DATE:

 January 30th 2016

DESCRIPTION:

 Calculate the spectral centroid values for each set in a set-palette. This
 will set the centroid slot of each set.

ARGUMENTS:

 - a set-palette object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - sort: If T, sorts the list values from high to low, otherwise the list
   will be returned with the order of the sets in the palette. Default =
   NIL.
 see this method in the chord class for a description of further keywords

RETURN VALUE:

 a list of two-element lists: the reference to the set within the palette
 and the centroid value.

SYNOPSIS:

```
(defmethod calculate-spectral-centroid ((sp set-palette)
                                        &rest keyargs &key &allow-other-keys)
```

28.5.315   set-palette/cmn-display

[set-palette] [Methods]

DESCRIPTION:

Generate printable music notation output (.EPS) of the given set-palette
object, including separate notation of the SUBSETS and RELATED-SETS slots,
using the Common Music Notation (CMN) interface. The method requires at
least the name of the given set-palette object, but has several additional
optional arguments for customizing output.

NB: Some of the keyword arguments are CMN attributes and share the same
    name as the CMN feature they effect.

ARGUMENTS:

- A set-palette object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :file. The file path, including the file name, of the file to be
  generated.
- :4stave. T or NIL to indicate whether the note-heads of the output should
  be printed on 4 staves (or 2). T = 4. Default = NIL.
- :text-x-offset. Number (positive or negative) to indicate the horizontal
  offset of any text in the output. A value of 0.0 results in all text
  being lined up left-flush with the note-heads below it. Units here and
  below are relative to CMN staff size. Default = -0.5.
- :text-y-offset. Number (positive or negative) to indicate the vertical
  offset of any text in the output.
- :font-size. A number indicating the size of any text font used in the
  output. This affects text only and not the music (see :size below for
  changing the size of the music). If 0 then no text will be displayed.
- :break-line-each-set. T or NIL to indicate whether each set-palette
  object should be printed on a separate staff or consecutively on the same
  staff. T = one staff per set-palette object. Default = T.
- :line-separation. A number to indicate the amount of white space between
  lines of music (systems), measured as a factor of the staff

height. Default = 3. This is a direct CMN attribute.
- :staff-separation. A number to indicate the amount of white space between
  staves belong to the same system, measured as a factor of the staff
  height. Default = 3. This is a direct CMN attribute.
- :transposition. Nil or a number (positive or negative) to indicate the
  number of semitones by which the pitches of the given set-palette object
  should be transposed before generating the CMN output. Default = NIL (0).
- :size. A number to indicate the size of the music-font in the CMN
  output. This affects music only, not text.
- :use-octave-signs. T or NIL to indicate whether to automatically insert
  ottava spanners. Automatic placement depends on the overall pitch
  content. This is a slippery-chicken process and may produce different
  results than :automatic-octave-signs, which is a direct CMN process.
  T = insert octave signs. Default = NIL.
- :automatic-octave-signs. T or NIL to indicate whether to automatically
  insert ottava spanners. Automatic placement depends on the overall pitch
  content. This is a direct CMN process and may produce different results
  than :use-octave-signs, which is a slippery-chicken process. T = insert
  octave signs. Default = NIL.
- :include-missing-chromatic. T or NIL to indicate whether to also print
  any chromatic pitches from the complete-set that are not present in the
  given set-palette object. T = print. Default = T.
- :include-missing-non-chromatic. T or NIL to indicate whether to also
  print any microtonal pitches from the complete-set that are not
  present in the given set-palette object. T = print. Default = T.

RETURN VALUE:

 T

EXAMPLE:

```
;; A typical example with some specified keyword values for file, font-size,
;; break-line-each-set, size, include-missing-chromatic and
;; include-missing-non-chromatic
(let ((msp (make-set-palette
            'test
            '((1 ((1
                   ((c3 g3 cs4 e4 fs4 a4 bf4 c5 d5 f5 gf5 af5 ef6)))
                  (2
                   ((c3 g3 cs4 e4 fs4 a4 bf4 c5 d5 f5 gf5 af5 ef6)
                    :subsets
                    ((tc1 (d2 g3 cs5))
                     (tc2 (eqs2 f3 bqf3))
                     (tc3 (b2 bqs3 gqf5)))))))
              (2 ((1 ((1 1) :transposition 5))
```

```
                        (2 ((1 2) :transposition 5))))
                  (3 ((1 ((1 1) :transposition -2))
                      (2 ((1 2) :transposition -2)))))))))
  (cmn-display msp
                :file "/tmp/sp-output.eps"
                :font-size 8
                :break-line-each-set nil
                :size 10
                :include-missing-chromatic nil
                :include-missing-non-chromatic nil))
```

SYNOPSIS:

```
(defmethod cmn-display ((sp set-palette) &rest keyargs &key &allow-other-keys)
```

28.5.316   set-palette/find-sets-with-pitches

[set-palette] [Methods]

DESCRIPTION:

```
Return a list of sets (as complete-set objects) from a given set-palette
object based on whether they contain specified pitches.

NB: Only sets which contain all of the specified pitches will be returned.
```

ARGUMENTS:

```
- A set-palette object.
- A list of pitches, either as pitch objects or note-name symbols.

OPTION ARGUMENTS
- T or NIL to indicate whether to print the notes of each successful set as
  they are being examined.
```

RETURN VALUE:

```
A list of complete-set objects.
```

EXAMPLE:

```
;; Find sets that contain a single pitch
(let ((msp (make-set-palette
             'test
             '((1 ((1
```

```
                    ((g3 c4 e4 g4)))
                   (2
                    ((c4 d4 e4 g4)))))
                (2 ((1 ((1 1) :transposition 5))
                    (2 ((1 2) :transposition 5))))
                (3 ((1 ((1 1) :transposition -2))
                    (2 ((1 2) :transposition -2)))))))))
  (find-sets-with-pitches msp '(c4)))

=>
(
COMPLETE-SET: complete: NIL
[...]
data: (BF3 C4 D4 F4)
[...]
COMPLETE-SET: complete: NIL
[...]
data: (C4 F4 A4 C5)
[...]
COMPLETE-SET: complete: NIL
[...]
data: (C4 D4 E4 G4)
[...]
COMPLETE-SET: complete: NIL
[...]
data: (G3 C4 E4 G4)
)

;; Search for a set of two pitches, printing the successfully matched sets
(let ((msp (make-set-palette
             'test
             '((1 ((1
                    ((g3 c4 e4 g4)))
                   (2
                    ((c4 d4 e4 g4)))))
                (2 ((1 ((1 1) :transposition 5))
                    (2 ((1 2) :transposition 5))))
                (3 ((1 ((1 1) :transposition -2))
                    (2 ((1 2) :transposition -2)))))))))
  (print (find-sets-with-pitches msp '(c4 f4) t)))

=>
(2 1): (C4 F4 A4 C5)
(3 2): (BF3 C4 D4 F4)
(
COMPLETE-SET: complete: NIL
```

```
[...]
data: (BF3 C4 D4 F4)
COMPLETE-SET: complete: NIL
[...]
data: (C4 F4 A4 C5)
)
```

SYNOPSIS:

```
(defmethod find-sets-with-pitches ((sp set-palette) pitches &optional print)
```

28.5.317   set-palette/force-micro-tone

[set-palette] [Methods]

DESCRIPTION:

Change the value of the MICRO-TONE slot of all pitch objects in a given
set-palette object to the specified <value>. NB If the pitches are
microtonal and thus have associated pitch-bends and microtonal frequencies,
these will not be changed, i.e. only the micro-tone slot is changed.

ARGUMENTS:

 - A set-palette object.

OPTIONAL ARGUMENTS:

 - An item of any type that is to be the new value of the MICRO-TONE slot of
   all pitch objects in the given sc-set object (generally T or
   NIL). Default = NIL.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;; Create a set-palette object whose individual sets contain some micro-tones
;; and print the contents of all the MICRO-TONE slots to see the values. Then
;; apply the force-micro-tone method and print the slots again to see the
;; changes.

(let ((msp (make-set-palette
```

```
                'test
                '((1 ((1
                       ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5 dqf5 gs5
                             b5)))
                      (2
                       ((bf1 d2 fqf2 fqs2 b2 c3 f3 g3 bqf3 bqs3 fs4 gs4 a4 cs5 gqf5)
                        :subsets
                        ((tc1 (d2 g3 cs5))
                         (tc2 (eqs2 f3 bqf3))
                         (tc3 (b2 bqs3 gqf5)))))))
                  (2 ((1 ((1 1) :transposition 5))
                      (2 ((1 2) :transposition 5))))
                  (3 ((1 ((1 1) :transposition -2))
                      (2 ((1 2) :transposition -2)))))))))
  (print (loop for i in (data msp)
            collect (loop for j in (data (data i))
                      collect (loop for p in (data j)
                                collect (micro-tone p)))))
  (force-micro-tone msp t)
  (print (loop for i in (data msp)
            collect (loop for j in (data (data i))
                      collect (loop for p in (data j)
                                collect (micro-tone p))))))

=>
(((NIL NIL T NIL NIL T T NIL NIL NIL NIL T T NIL NIL)
  (NIL NIL T T NIL NIL NIL NIL T T NIL NIL NIL NIL T))
 ((NIL NIL T NIL NIL T T NIL NIL NIL NIL T T NIL NIL)
  (NIL NIL T T NIL NIL NIL NIL T T NIL NIL NIL NIL T))
 ((NIL NIL T NIL NIL T T NIL NIL NIL NIL T T NIL NIL)
  (NIL NIL T T NIL NIL NIL NIL T T NIL NIL NIL NIL T)))

(((T T T T T T T T T T T T T T T) (T T T T T T T T T T T T T T T))
 ((T T T T T T T T T T T T T T T) (T T T T T T T T T T T T T T T))
 ((T T T T T T T T T T T T T T T) (T T T T T T T T T T T T T T T)))
```

SYNOPSIS:

```
(defmethod force-micro-tone ((sp set-palette) &optional value)
```

28.5.318   set-palette/gen-max-coll-file

[set-palette] [Methods]

DATE:

26-Dec-2009

DESCRIPTION:

Write a text file from a given set-palette object suitable for reading into
Max/MSP's coll object. The resulting text file has one line for each set in
the palette, with the coll index being the ID of the set. The rest of the
line is a list of frequency/amplitude pairs (or MIDI note numbers if
required).

ARGUMENTS:

 - A set-palette object.
 - The name (and path) of the .txt file to write.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether MIDI note numbers or frequencies should be
   generated. T = MIDI. Default = NIL (frequencies).

RETURN VALUE: EXAMPLE:

```
;; Generates frequencies by default
(let ((msp (make-set-palette
            'test
            '((1 ((1
                   ((g3 c4 e4 g4)))
                  (2
                   ((c4 d4 e4 g4)))))
              (2 ((1 ((1 1) :transposition 5))
                  (2 ((1 2) :transposition 5))))
              (3 ((1 ((1 1) :transposition -2))
                  (2 ((1 2) :transposition -2)))))))))
  (gen-max-coll-file msp "/tmp/msp-mcf.txt"))

;; Set the optional argument to T to generate MIDI note numbers instead
(let ((msp (make-set-palette
            'test
            '((1 ((1
                   ((g3 c4 e4 g4)))
                  (2
                   ((c4 d4 e4 g4)))))
              (2 ((1 ((1 1) :transposition 5))
                  (2 ((1 2) :transposition 5))))
              (3 ((1 ((1 1) :transposition -2))
```

```
                    (2 ((1 2) :transposition -2))))))))
  (gen-max-coll-file msp "/tmp/msp-mcf.txt" t))
```

SYNOPSIS:

```
(defmethod gen-max-coll-file ((sp set-palette) file &optional midi)
```

28.5.319   set-palette/gen-midi-chord-seq

[set-palette] [Methods]

DESCRIPTION:

Generate a MIDI file in which each set of the given set-palette object is
played at 1 second intervals.

ARGUMENTS:

 - A set-palette object.
 - The name and path for the MIDI file to be generated.

RETURN VALUE:

 Always returns T

EXAMPLE:

```
(let ((msp (make-set-palette
            'test
            '((1 ((1
                   ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5 dqf5 gs5
                       b5)))
                 (2
                  ((bf1 d2 fqf2 fqs2 b2 c3 f3 g3 bqf3 bqs3 fs4 gs4 a4 cs5 gqf5)
                   :subsets
                   ((tc1 (d2 g3 cs5))
                    (tc2 (eqs2 f3 bqf3))
                    (tc3 (b2 bqs3 gqf5)))))))
             (2 ((1 ((1 1) :transposition 5))
                 (2 ((1 2) :transposition 5))))
             (3 ((1 ((1 1) :transposition -2))
                 (2 ((1 2) :transposition -2))))))))
  (gen-midi-chord-seq msp "/tmp/msp-gmchs.mid"))
```

SYNOPSIS:

```
(defmethod gen-midi-chord-seq ((sp set-palette) midi-file
                               &optional (tempo 60.0))
```

28.5.320   set-palette/limit

[set-palette] [Methods]

DESCRIPTION:

 Limit all the sets in a palette to a specified range. See the tl-set class
 limit method for details of the this method and its arguments.

RETURN VALUE:

 The set-palette object.

SYNOPSIS:

```
(defmethod limit ((sp set-palette) &key upper lower do-related-sets)
```

28.5.321   set-palette/limit-shift-octave

[set-palette] [Methods]

DATE:

 19th August 2015, Edinburgh

DESCRIPTION:

 Apply the tl-set method of the same name to each set in the palette. See
 that class method description for details.

ARGUMENTS:

 - the set-palette object

OPTIONAL ARGUMENTS:

 See descriptions in tl-set limiti-shift-octave method noting that here the
 :upper and :lower arguments can be a single note symbol or pitch object, as
 in tl-set, or an envelope. In the latter case, the x axis can be over any
 arbitrary scale and the y values can either be note symbols or midi note
 numbers.

RETURN VALUE:

 The set-palette object (modified).

SYNOPSIS:

```
(defmethod limit-shift-octave ((sp set-palette) &key upper lower
                                                  do-related-sets)
```

28.5.322 set-palette/make-set-palette

[set-palette] [Functions]

DESCRIPTION:

 Create a set-palette object.

 Note that the sets in this palette may refer to previously defined sets in
 order to avoid retyping note lists (see example below).

ARGUMENTS:

 - A symbol that is to be the ID of the resulting set-palette object.
 - A recursive list of key/data pairs, of which the deepest level of data
   will be a list of note-name symbols.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :recurse-simple-data. T or NIL to indicate whether to interpret
   two-element data lists as recursive palettes. Default = T.
 - :warn-note-found. T or NIL to indicate whether to print warnings when
   specified data is not found with subsequent calls to the get-data method.

RETURN VALUE:

 A set-palette object.

EXAMPLE:

```
;;; Create a set-palette object
(make-set-palette
 'test
 '((1 ((1
```

```
          ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5 dqf5 gs5 b5)
           :subsets
           ((tc1 ((ds2 e3 a4) "a-tag"))
            (tc2 (bf1 d4 cqs5))
            (tc3 (c3 cs4 gs5)))))
        (2
         ((bf1 d2 fqf2 fqs2 b2 c3 f3 g3 bqf3 bqs3 fs4 gs4 a4 cs5 gqf5)
          :subsets
          ((tc1 (d2 g3 cs5))
           (tc2 (eqs2 f3 bqf3))
           (tc3 (b2 bqs3 gqf5)))))
        (3
         ((cqs2 fs2 g2 c3 d3 fqs3 gqf3 cs4 ds4 e4 gs4 dqf5 f5 a5 bqs5)
          :subsets
          ((tc1 (cqs2 c3 f5))
           (tc2 (fs2 e4 bqs5))
           (tc3 (d3 ef4 a5)))))))
    (2 ((1 ((1 1) :transposition 5))
        (2 ((1 2) :transposition 5))
        (3 ((1 3) :transposition 5))))
    (3 ((1 ((1 1) :transposition -2))
        (2 ((1 2) :transposition -2))
        (3 ((1 3) :transposition -2)))))))

=>
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 9
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: TEST, tag: NIL,
data: (
[...]

;;; NB A simple list of sets (with unique id slots) can also be passed.

;;; Create a set-palette object by referencing a set-palette-object already
;;; defined (sp1) and transposing a clone of that object.
(make-set-palette
 'test
 '((sp1
```

```
   ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5
         dqf5 gs5 b5)
    :subsets
    ((tc1 (ds2 e3 a4))
     (tc2 (bf1 d4 cqs5))
     (qc1 (aqf2 e3 a4 dqf5 b5))
     (qc2 (bf1 c3 gqs3 cs4 cqs5)))
    :related-sets
    ((missing (bqs0 eqs1 f5 aqs5 eqf6 fqs6
                    bqf6 dqs7 fs7)))))
  (sp2
   (sp1 :transposition 13))))

=>
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 2
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: TEST, tag: NIL,
data: (
COMPLETE-SET: complete: NIL
              num-missing-non-chromatic: 7
              num-missing-chromatic: 2
              missing-non-chromatic: (BQS BQF AQS FQS EQS EQF DQS)
              missing-chromatic: (FS F)
[...]
    subsets:
TC1: (DS2 E3 A4)
TC2: (BF1 D4 CQS5)
QC1: (AQF2 E3 A4 DQF5 B5)
QC2: (BF1 C3 GQS3 CS4 CQS5)
    related-sets:
MISSING: (BQS0 EQS1 F5 AQS5 EQF6 FQS6 BQF6 DQS7 FS7)
SCLIST: sclist-length: 15, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SP1, tag: NIL,
data: (BF1 EF2 AQF2 C3 E3 GQF3 GQS3 CS4 D4 G4 A4 CQS5 DQF5 GS5 B5)
[...]
COMPLETE-SET: complete: NIL
              num-missing-non-chromatic: 7
```

```
                num-missing-chromatic: 2
                missing-non-chromatic: (BQS BQF AQS FQS EQS EQF DQS)
                missing-chromatic: (FS F)
TL-SET: transposition: 13
        limit-upper: NIL
        limit-lower: NIL
[...]
    subsets:
TC1: (E3 F4 BF5)
TC2: (B2 EF5 DQF6)
QC1: (AQS3 F4 BF5 DQS6 C7)
QC2: (B2 CS4 AQF4 D5 DQF6)
    related-sets:
MISSING: (BQS0 EQS1 F5 AQS5 EQF6 FQS6 BQF6 DQS7 FS7)
SCLIST: sclist-length: 15, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SP2, tag: NIL,
data: (B2 E3 AQS3 CS4 F4 GQS4 AQF4 D5 EF5 AF5 BF5 DQF6 DQS6 A6 C7)
**************
)
```

SYNOPSIS:

```
(defun make-set-palette (id palette
                          &key (recurse-simple-data t) (warn-not-found t))
```

28.5.323   set-palette/midi-play

[set-palette] [Methods]

DATE:

 25th May 2017, Edinburgh

DESCRIPTION:

 Write a MIDI file containing the sets in the set-palette

ARGUMENTS:

 - the set-palette object

OPTIONAL ARGUMENTS:

 keyword argument:

- :midi-file. The path (string) of the midi file to write. Default is to
use the ID of the set-palette and write it into the default directory.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod midi-play ((sp set-palette)
                      &key
                        (tempo 60.0)
                        (auto-open (get-sc-config 'midi-play-auto-open))
                        (midi-file
                         (format nil "~a~a.mid"
                                 (get-sc-config 'default-dir)
                                 (string-downcase (string (id sp)))))))
```

28.5.324   set-palette/quality-extremes

[set-palette] [Methods]

DESCRIPTION:

 Get the extremes (as four values) of the dissonance (min max) and spectral
 centroid (min max) of a whole set palette.

ARGUMENTS:

 - a set-palette object

RETURN VALUE:

 Four values (i.e. to be used by multiple-value-bind and friends):
 dissonance min, dissonance max, centroid min, centroid max.

SYNOPSIS:

```
(defmethod quality-extremes ((sp set-palette))
```

28.5.325   set-palette/recursive-set-palette-from-ring-mod

[set-palette] [Functions]

DESCRIPTION:

Create a set-palette object consisting of sub palette-objects whose pitch
content is generated based on ring modulation routines applied to the
specified pitches.

ARGUMENTS:

- A list of note-name symbols, each of which will serve as the reference
  pitch from which a new set-palette object is made using the
  set-palette-from-ring-mod method.
- A symbol that will be the ID for the top-level set-palette object. The
  IDs of the new set-palette objects contained in the top-level object are
  generated from the note-name symbols of the reference-pitches, with the
  IDs of the pitch sets contained with them then generated by sequential
  numbers.

OPTIONAL ARGUMENTS:

keyword arguments:
- :partials. A list of integers that are the partials which the method is
  to ring modulate, with 1 being either the reference-note or the bass note
  that would have the reference-note as the highest partial in the given
  list. Default = '(1 3 5 7).
- :warn-no-bass. T or NIL to indicate whether to issue a warning when
  ring-mod-bass fails to find suitable bass notes for the generated sets. T
  = warn. Default = T.
- :do-bass. T or NIL to indicate whether to add notes created by the
  ring-mod-bass function to the resulting set-palette object. T = create and
  add bass notes. Default = T.
- :remove-octaves. T or NIL to indicate whether to remove the upper
  instances of any octave-equivalent pitches from the resulting set-palette
  object. T = remove. Default = NIL.
- :min-bass-notes. An integer that is the minimum number of bass notes to
  be generated and added to the resulting set-palette object. Default = 1.
- :ring-mod-bass-octave. An integer that is the MIDI octave reference
  number (such as the 4 in 'C4), indicating the octave from which the bass
  note(s) are to be taken.
- :force-chromatic. T or NIL. If T, force all micro-tone slots of pitch objects
  to be NIL so that they won't be filtered out when a set is to be used by a
  chromatic instrument. See sc-set class force-micro-tone for more details.

RETURN VALUE:

- A set-palette object (recursive)

EXAMPLE:

```
;; Simple useage with default keyword argument values
(recursive-set-palette-from-ring-mod '(a4 b4 c4) 'rspfrm-test)

=>
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 3
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: RSPFRM-TEST, tag: NIL,
data: (
NAMED-OBJECT: id: A4, tag: NIL,
data:
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 21
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 21, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: A4, tag: NIL,
data: (
COMPLETE-SET: complete: NIL
[...]
```

SYNOPSIS:

```
(defun recursive-set-palette-from-ring-mod (reference-notes id
                                            &key
                                               (warn-no-bass t)
                                               (ring-mod-bass-octave 0)
                                               (do-bass t)
                                               remove-octaves
                                               force-chromatic
                                               (min-bass-notes 1)
                                               (partials '(1 3 5 7)))
```

28.5.326   set-palette/remove-similar

[set-palette] [Methods]

DATE:

 12th August 2015, Wals, Austria

DESCRIPTION:

 Remove similar sets from a palette when two sets are deemed
 similar. The set furthest down the data list will remain, whatever the
 ID. Note that this is a destructive operation.

 We use the similarity chord method to remove sets from the palette, so see
 that method for further information and for more keyword argument
 descriptions.

ARGUMENTS:

 - the set-palette object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :threshold. The lowest value that the chord class's similarity method may
    return in order to trigger removal. Default = 0.8
 - :clone. Whether to clone the set-palette object before removing sets.
 - :enharmonics-are-equal. See similarity method. Default = T.
 - :octaves-are-true. See similarity method. Default = NIL.

RETURN VALUE:

 the pared-down set-palette object

SYNOPSIS:

```
(defmethod remove-similar ((sp set-palette) &key (threshold 0.8)
                                            (clone nil)
                                            (enharmonics-are-equal t)
                                            (octaves-are-true nil))
```

28.5.327   set-palette/ring-mod

[set-palette] [Functions]

DESCRIPTION:

Ring modulate (sum and difference tones) two pitches and return the
resulting pitch and harmonic partials thereof.

ARGUMENTS:

- A first pitch, either as a numeric hertz frequencey or a note-name
  symbol.
- A second pitch, either as a numeric hertz frequencey or a note-name
  symbol. The second value needn't be higher than first.

OPTIONAL ARGUMENTS:

keyword arguments
- :return-notes. T or NIL to indicate whether to return the results as
  note-name symbols or frequency numbers. T = note-name symbols.
  Default = NIL.
- :pitch1-partials. An integer that indicates how many harmonic partials of
  the first pitch are to be included in the modulation. Default = 3.
- :pitch2-partials. An integer that indicates how many harmonic partials of
  the second pitch are to be included in the modulation. Default = 2.
- :min-freq. A number that is the the minimum frequency (hertz) that may be
  returned. Default = 20.
- :max-freq. A number that is the the maximum frequency (hertz) that may be
  returned. Default = 20000.
- :round. T or NIL to indicate whether frequency values returned are first
  rounded to the nearest hertz. T = round. Default = T
- :remove-duplicates. T or NIL to indicate whether any duplicate
  frequencies are to be removed from the resulting list before returning
  it. T = remove. Default = T.
- :print. T or NIL to indicate whether resulting data is to be printed as
  it is being generated. T = print. Default = NIL.
- :remove-octaves. T or NIL to indicate whether octave repetitions of
  pitches will be removed from the resulting list before returning it,
  keeping only the lowest instance of each pitch. This argument can also be
  set as a number or a list of numbers that indicates which octave
  repetitions will be allowed, the rest being removed. For example,
  :remove-octaves '(1 2) will remove all octave repetitions of a given
  pitch except for those that are 1 octave and 2 octaves above the given
  pitch; thus '(c1 c2 c3 c4 c5) would return '(c1 c2 c3), removing c4 and
  c5. Default = NIL.
- :scale. A variable that indicates which scale to use when converting
  frequencies to note-names. Default = cm::*scale* i.e. the value to which
  the Common Music scale is set, which in slippery chicken is
  quarter-tone by default.

RETURN VALUE:

 A list of note-name symbols or frequencies.

EXAMPLE:

```
;; Apply ring modulation to 'C4 and 'D4, using 5 partials of the first pitch
;; and 3 partials of the second, removing octave repetitions, and returning the
;; results as rounded hertz-frequencies
(ring-mod 'c4 'd4
          :pitch1-partials 5
          :pitch2-partials 3
          :min-freq 60
          :max-freq 2000
          :remove-octaves t)

=> (64.0 96.0 166.0 198.0 230.0 358.0 427.0 459.0 491.0 555.0 619.0 817.0
    1079.0 1143.0 1340.0 1372.0 1404.0 1666.0 1895.0 1927.0)

;; Applying ring modulation to two frequencies, returning the results as
;; note-name symbols within the chromatic scale.
(ring-mod '261.63 '293.66
          :return-notes t
          :remove-duplicates nil
          :scale cm::*chromatic-scale*)

=> (C1 C2 G3 BF3 E4 B4 CS5 AF5 AF5 CS6 CS6 F6)
```

SYNOPSIS:

```
(defun ring-mod (pitch1 pitch2 ;; hertz or notes
                 &key (return-notes nil) (pitch1-partials 3) (pitch2-partials 2)
                 (min-freq 20) (max-freq 20000) (round t) (remove-duplicates t)
                 (print nil) remove-octaves (scale cm::*scale*))
```

28.5.328   set-palette/ring-mod-bass

[set-palette] [Functions]

DESCRIPTION:

 Using ring-modulation techniques, invent (sensible) bass note(s) from a
 list of frequencies.

ARGUMENTS:

 - A list of numbers that are hertz frequencies from which the bass note(s)
   are to be generated.

OPTIONAL ARGUMENTS:

```
keyword arguments
- :bass-octave. An integer that is an octave indicator (e.g. the 4 in
  'C4). The method will only return any frequencies/note-names generated
  that fall in this octave. Default = 0.
- :low. A note-name symbol that is the lowest possible pitch of those
  returned. This argument further restricts the :bass-octave argument. Thus
  a :bass-octave value of 1 could be further limited to no pitches below
  :low 'DS1. Default = 'A0.
- :high. A note-name symbol that is the highest possible pitch of those
  returned. This argument further restricts the :bass-octave argument. Thus
  a :bass-octave value of 1 could be further limted to no pitches above
  :high 'FS1. Default = 'G3.
- :round. T or NIL to indicate whether the frequencies returned are rounded
  to integer values. T = round. Default = T.
- :warn. T or NIL to print a warning when no bass can be created from the
  specified frequencies/note-names. T = print warning. Default = T.
- :return-notes. T or NIL to indicate whether the resulting pitches should
  be returned as note-names instead of frequencies. T = return as
  note-names. Default = NIL.
- :scale. A variable pointing to the scale to which any translation of
  frequencies into note-names symbols should take place. By default this
  value is set to cm::*scale*, which is automatically set by slippery
  chicken to 'quarter-tone at initialisation. To return e.g. pitches rounded
  to chromatic note-names set this argument to cm::*chromatic-scale*.
```

RETURN VALUE:

```
 Returns a list of frequencies by default.

 Setting the :return-notes keyword argument to T will cause the method to
 return note-name symbols instead.
```

EXAMPLE:

```
;; Simple usage with default keyword argument values
(ring-mod-bass '(261.63 293.66 329.63 349.23))

=> (28 29 32)

;; Return as note-names instead, in quarter-tone scale by default
(ring-mod-bass '(261.63 293.66 329.63 349.23)
               :return-notes t)
```

```
=> (A0 BF0 BQS0)

;; Set the :scale argument to cm::*chromatic-scale* to return equal-tempered
;; note-name symbols instead
(ring-mod-bass '(261.63 293.66 329.63 349.23)
               :return-notes t
               :scale cm::*chromatic-scale*)

=> (A0 BF0 C1)

;; Return pitches from bass octave 1 rather than default 0
(ring-mod-bass '(261.63 293.66 329.63 349.23 392.00)
               :return-notes t
               :scale cm::*chromatic-scale*
               :bass-octave 1)

=> (CS1 D1 F1 G1 A1 B1)

;; Further limit the notes returned by setting :low and :high values
(ring-mod-bass '(261.63 293.66 329.63 349.23 392.00)
               :return-notes t
               :scale cm::*chromatic-scale*
               :bass-octave 1
               :low 'e1
               :high 'a1)

=> (F1 G1)

;; Set the :round argument to NIL to return decimal-point frequencies
(ring-mod-bass '(261.63 293.66 329.63 349.23 392.00)
               :bass-octave 1
               :low 'e1
               :high 'a1
               :round NIL)

=> (42.76999999999998 43.45666666666667 43.80000000000001 49.16999999999999)

;; The method prints a warning by default if no bass note can be made
(ring-mod-bass '(261.63))

=>
NIL
WARNING: set-palette::ring-mod-bass: can't get bass from (261.63)!

;; This warning can be suppressed by setting the :warn argument to NIL
(ring-mod-bass '(261.63) :warn nil)
```

```
=> NIL
```

SYNOPSIS:

```
(defun ring-mod-bass (freqs &key (bass-octave 0) (low 'a0) (high 'g3) (round t)
                      (warn t) (return-notes nil) (scale cm::*scale*))
```

28.5.329   set-palette/round-to-nearest

[set-palette] [Methods]

DATE:

 January 30th 2017

DESCRIPTION:

 Rounds all pitch objects to the nearest pitch in the current or given
 scale. See also sc-set, chord, slippery-chicken, event, and pitch class
 methods.

ARGUMENTS:

 - a set-palette object

OPTIONAL ARGUMENTS:

 keyword argument:
 :scale. The scale to use when rounding. (Common Music tuning object or
 symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or 'quarter-tone
 only at present. Default is the current scale as set by (in-scale :...).

RETURN VALUE:

 the modified set-palette

SYNOPSIS:

```
(defmethod round-to-nearest ((sp set-palette) &key (scale cm::*scale*))
```

28.5.330   set-palette/set-palette-from-ring-mod

[set-palette] [Functions]

DESCRIPTION:

Create a new set-palette object from the pitches returned by applying ring
modulation procedures (difference and sum tones of partials).

ARGUMENTS:

- A note-name symbol that is the central pitch from which we perform the
  ring-modulation. See :partials below.
- A symbol that is to be the ID for the new set-palette object.

OPTIONAL ARGUMENTS:

keyword arguments
- :partials. A list of integers that are the partials which the method uses
  to ring modulate. We create partials ascending from the reference-note
  but also ascending from a fundamental calculated so that reference-note
  would be the highest partial in the partials list. E.g. if
  reference-note were 'a4 (440Hz) and :partials was '(1 2) we'd have
  partial frequencies of 440 and 880, as these are the ascending partials 1
  and 2 from 440, but also have 220, as that is the fundamental for which
  440 would be the highest partial out of (1 2). Default = '(1 3 5 7).
- :warn-no-bass. T or NIL to indicate whether to issue a warning when
  ring-mod-bass fails to find suitable bass notes for the generated sets.
  T = warn. Default = T.
- :do-bass. T or NIL to indicate whether to add notes created by the
  ring-mod-bass function to the resulting set-palette object. T = create and
  add bass notes. Default = T.
- :remove-octaves. T or NIL to indicate whether to remove the upper
  instances of any octave-equivalent pitches from the resulting set-palette
  object. T = remove. Default = NIL.
- :min-bass-notes. An integer that is the minimum number of bass notes to
  be generated and added to the resulting set-palette object. Default = 1.
- :ring-mod-bass-octave. An integer that is the MIDI octave reference
  number (such as the 4 in 'C4), indicating the octave from which the bass
  note(s) are to be taken.
- :force-chromatic. T or NIL. If T, force all micro-tone slots of pitch
  objects to be NIL so that they won't be filtered out when a set is to be
  used by a chromatic instrument. See sc-set class force-micro-tone for
  more details. If you want truly chromatic sets make sure to be (in-scale
  :chromatic)
- :start-id. An integer which represents the ID number for the first
  set. This will be incremented for each subsequent set generated.

RETURN VALUE:

A set-palette object.

EXAMPLE:

```
;; Simple usage with default keyword argument values
(set-palette-from-ring-mod 'a4 'spfrm-test)

=>
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 21
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 21, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SPFRM-TEST, tag: NIL,
data: (
[...]

;;; Use with the :partials argument
(let ((spfrm2 (set-palette-from-ring-mod 'a4 'spfrm-test
                                         :partials '(2 4 6 8))))
  (loop for cs in (data spfrm2) collect (pitch-symbols cs)))

=> ((BQS0 CS5 E5 GQF5 B5 CS6 DQS6 FQS6 GQF6 AF6 BF6 B6 C7)
    (BQF0 B0 A2 A3 E4 A4 CS5 E5 GQF5)
    (BQS0 FQS6 GQF6 AF6 BF6 B6 C7 GQS7 AF7 AQF7 AQS7 BF7 BQF7)
    (B0 A2 A3 E4 A4 CS5 E5 GQF5 A5 B5) (BQS0 DQF7 DQS7 EQF7 EQS7 FQS7 FS7)
    (BQF0 A2 A3 E4 CS5 E5 GQF5 B5 CS6 DQS6)
    (AQS0 BQF0 B0 GQS7 AF7 AQF7 AQS7 BF7 BQF7)
    (B0 A4 E5 CS6 E6 GQF6 B6 CS7 DQS7 FQS7 GQF7) (B0 A5 A6 E7 A7)
    (B0 A3 CS5 CS6 DQS6 FQS6 GQF6 B6 C7 DQF7 DQS7 FS7 AF7) (BQS0 A5 A6 E7 A7)
    (B0 A4 A5 E6 A6 CS7 E7 GQF7 A7) (BQS0 A5 A6 E7)
    (BQS0 CS6 E6 GQF6 B6 CS7 DQS7 FQS7 GQF7 AF7 BF7 B7 C8)
    (BQF0 B0 BQS0 A2 A3 E4 A4 CS5 GQF5 A5 B5 CS6 E6)
    (BQS0 B6 CS7 DQS7 FQS7 GQF7 AF7) (B0 A3 A4 E5 A5 CS6 E6 GQF6)
    (B0 BQS0 FQS7 GQF7 AF7 BF7 B7 C8) (BQS0 CS6 FQS6 C7 DQS7 FQS7 GQS7 BQF7 C8)
    (B0 A5 A6 E7 A7) (BQS0 A5 E6 CS7 E7 GQF7 B7) (BQS0 A6 A7)
    (BQS0 AF6 B6 DQF7 FS7 AF7 AQS7)
    (BQF0 B0 BQS0 A2 A3 CS5 GQF5 CS6 DQS6 FQS6 GQF6 BF6)
    (BQS0 EQF7 FQS7 GQS7 BQF7 C8) (BQS0 A6 CS7 GQF7 A7) (B0 A5 A6)
    (BQS0 CS7 E7 GQF7 B7))

;;; Use with the :do-bass and :remove-octaves arguments
(let ((spfrm3 (set-palette-from-ring-mod 'a4 'spfrm-test
```

```
                                          :do-bass nil
                                          :remove-octaves t)))
  (loop for cs in (data spfrm3) collect (pitch-symbols cs)))
=> ((BQS1 GQF3 EF4 A4 DQF5) (DQF6 DQS6 EF6 F6 FQS6 GQF6 EQF7 EQS7)
    (BQS2 GQF3 A4 DQF5 F5 GQS5) (BQS6 C7 CQS7 DQF7 D7 DQS7)
    (BQS3 EF4 GQF4 DQF5 F5 GQS5 BF5 CQS6) (FQS7 FS7 GQF7 G7 GQS7)
    (GQF5 BF5 DQF6 AQF6 C7 DQS7 F7 GQS7) (BQS1 A4 F5 GQS5 DQS6)
    (GQS6 AQS6 BQS6 DQS7 EQF7 F7 BQF7) (BQS1 EF4 F5 GQS5 CQS6 FQS6)
    (EF7 EQS7 FQS7 GQS7 AQF7 AQS7) (F5 BQS5 GQS6 B6 D7 FS7 AF7 AQS7) (A4 GQF7)
    (A4 CS7 GQF7) (A4 CS7) (EF6 G6 BF6 F7 AQF7 BQS7)
    (BQS1 GQF3 DQF5 CQS6 DQS6 F6 AQS6) (CQS7 DQS7 F7 AQF7 BF7 BQS7)
    (E6 A6 GQF7 B7) (A4 E6 B7) (A6 CS7 E7 B7))
```

SYNOPSIS:

```
(defun set-palette-from-ring-mod (reference-note id
                                  &key
                                     (warn-no-bass t)
                                     (do-bass t)
                                     ;; the start id for the sets; will be
                                     ;; incremented
                                     (start-id 1)
                                     remove-octaves
                                     force-chromatic
                                     (min-bass-notes 1)
                                     (ring-mod-bass-octave 0)
                                     (partials '(1 3 5 7)))
```

28.5.331   set-palette/set-palette-from-spectra

[set-palette] [Functions]

DATE:

```
 August 24th 2018
```

DESCRIPTION:

```
 Create a set-palette by analysing a sound file at regular intervals and
 using the detected frequency components as pitches in a set.
```

ARGUMENTS:

```
 - the sndfile to analyse
```

OPTIONAL ARGUMENTS:

```
keyword arguments
- :id. The ID for the set-palette object to return. If NIL a suitable symbol
  will be created from the sndfile path. Default = NIL.
- :start. The time in seconds to start the analysis. Default = 0
- :stop. The time in seconds to stop the analysis. If nil then analysis will
  continue until one millisecond before the end of the sound file. Default =
  NIL.
- :interval. The time increment in seconds for the analysis start
  point. Default = 0.1
- :min-pitches. The minimum number of pitches a set may contain. Analyses
  returning less than this number of frequencies will be skipped. Default
  = 5.
- :pitch-min. The lowest pitch of any set. Pitch object or symbol. Default
  = c1.
- :pitch-max. The highest pitch of any set. Pitch object or symbol. Default
  = g7.
- :round. Round the generated pitches to the nearest in the current scale?
  Default = NIL.
- :reject-fun. A function that takes a set as an argument and returns T if
  it should be rejected, or NIL if not. Default = NIL.
- :warn-dups. T or NIL to warn if duplicate pitches are found when calling
  make-complete-set. Default = T.
- :gs-keys. Any other keyword arguments to be passed to the get-spectrum
  call (get-spectrum.lsp). In particular :fftsize will be useful here as
  larger windows will result in lower pitches being detected. In any case
  it's worth experimenting with :fftsize. Default: various but :fftsize of
  1024
```

RETURN VALUE:

```
a set-palette object. Each set in the palette has a numerically ascending
integer ID, starting from 1.
```

EXAMPLE:

```
(let ((sp (set-palette-from-spectra
            (concatenate 'string cl-user::+slippery-chicken-home-dir+
                         "test-suite/test-sndfiles-dir-1/test-sndfile-3.aiff"))))
  (print-simple sp))
=>
1: DQS2 E5 DQS6 AQF6 AQS6 B6 CQS7 DQF7
2: E5 GQS5 AQS5 B5 CQS6 AQS6
4: EQF3 FQS4 E5 AQS5 CS6 AQS6
5: EQS2 AF3 EQF4 A4 DQF5 E5 AQF5 BQF5 AQS6
6: B4 EQF5 BF5 CQS6 DQS6 EQS6 GQS6 AQS6
7: F4 A4 CQS5 EQF5 GQS5 BQS5 CS6 AQS6
```

```
23: E5 BF6 D7 EQF7 E7 FQS7
NIL
```

SYNOPSIS:

```
(defun set-palette-from-spectra (sndfile
                                 &key
                                   id
                                   (start 0)
                                   stop
                                   (interval 0.1)
                                   (min-pitches 5)
                                   (pitch-min 'c1)
                                   (pitch-max 'g7)
                                   (round t)
                                   reject-fun
                                   (warn-dups t)
                                   (gs-keys
                                    '(:fftsize 1024
                                                :order-by clm::freq
                                                :freq-min 20
                                                :freq-max ,(note-to-freq 'b8))))
```

28.5.332   set-palette/set-palette-p

[set-palette] [Functions]

DESCRIPTION:

 Test whether a given object is a set-palette object.

ARGUMENTS:

 - A lisp object

EXAMPLE:

```
(let ((msp (make-set-palette
            'test
            '((1 ((1
                   ((bf1 ef2 aqf2 c3 e3 gqf3 gqs3 cs4 d4 g4 a4 cqs5 dqf5 gs5
                     b5)))
                  (2
                   ((bf1 d2 fqf2 fqs2 b2 c3 f3 g3 bqf3 bqs3 fs4 gs4 a4 cs5 gqf5)
                    :subsets
```

```
                ((tc1 (d2 g3 cs5))
                 (tc2 (eqs2 f3 bqf3))
                 (tc3 (b2 bqs3 gqf5)))))))
           (2 ((1 ((1 1) :transposition 5))
               (2 ((1 2) :transposition 5))))
           (3 ((1 ((1 1) :transposition -2))
               (2 ((1 2) :transposition -2)))))))))
  (set-palette-p msp))

=> T
```

RETURN VALUE:

 t or nil

SYNOPSIS:

```
(defun set-palette-p (thing)
```

28.5.333   set-palette/stack

[set-palette] [Methods]
DATE:

 30th June 2019, Corfu

DESCRIPTION:

 Call the stack method for each set in the palette. See the method
 description in the sc-set class for details of arguments etc.

RETURN VALUE:

 - the set-palette with new stacked sets (clone before calling if necessary)

SYNOPSIS:

```
(defmethod stack ((sp set-palette) num-stacks &key id by-freq (up t) (down t))
```

28.5.334   set-palette/thin

[set-palette] [Methods]
DESCRIPTION:

 Apply the thin method to each set in the palette. See the sc-set class
 method for details, including keyword arguments.

SYNOPSIS:

(defmethod thin ((sp set-palette) &rest keywords &key &allow-other-keys)

28.5.335   set-palette/wrap

[set-palette] [Methods]

DATE:

 March 5th 2016, Edinburgh

DESCRIPTION:

 Applies the wrap method from the chord class to each set in the
 palette. See the chord class method for details.

RETURN VALUE:

 The set-palette object, wrapped.

SYNOPSIS:

(defmethod wrap ((sp set-palette) &optional (num-times 1) (transpose t))

28.5.336   palette/sndfile-palette

[palette] [Classes]

NAME:

 sndfile-palette

 File: sndfile-palette.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list ->
                  palette -> sndfile-palette

 Version: 1.0.11

```
      Project:            slippery chicken (algorithmic composition)

      Purpose:            Implementation of the sndfile-palette class, which is a
                          simple palette that checks that all the sound files given
                          in a list for each id exist.  See comments in methods for
                          limitations and special features of this class.

      Author:             Michael Edwards: m@michael-edwards.org

      Creation date:      18th March 2001

      $$ Last modified:  11:15:12 Wed Apr 15 2020 CEST

      SVN ID: $Id$
```

28.5.337   sndfile-palette/find-sndfile

[sndfile-palette] [Methods]

DESCRIPTION:

Return the full directory path and file name of a specified sound file,
from within the directories given in the PATHS slot.

ARGUMENTS:

- A sndfile-palette object.
- The name of a sound file from within that object. This can be a string or
  a symbol. Unless it is given as a string, it will be handled as a symbol
  and will be converted to lowercase. Inclusion of the file extension is
  optional.

RETURN VALUE:

Returns the full directory path and file name of the specified sound file
as a string.

EXAMPLE:

```
(let ((msfp (make-sfp 'sfp-test
                      '((sndfile-group-1
                         (test-sndfile-1))
                        (sndfile-group-2
                         (test-sndfile-2 test-sndfile-3
                          (test-sndfile-4 :frequency 261.61)))
```

```
                            (sndfile-group-3
                             ((test-sndfile-5 :start 0.006 :end 0.182)
                              test-sndfile-6)))
                          :paths
                          '("/path/to/sndfiles-dir-1"
                            "/path/to/sndfiles-dir-2"))))
 (find-sndfile msfp 'test-sndfile-4))

=> "/path/to/sndfiles-dir-2/test-sndfile-4.aiff"
```

SYNOPSIS:

```
(defmethod find-sndfile ((sfp sndfile-palette) sndfile)
```

28.5.338   sndfile-palette/get-nearest

[sndfile-palette] [Methods]

DESCRIPTION:

 Get the sndfile whose frequency slot is nearest to the first argument.

ARGUMENTS:

 - the frequency we're looking for, in Hertz (a number)
 - the sndfile-palette object we'll search

OPTIONAL ARGUMENTS:

 - the IDs of the sndfile groups we'll search. Either a list of ids or a
 single id. If NIL then all groups will be searched. Default = NIL.
 - if more than one sndfile is very close in frequency, select one at random
 (see also the get-nearest-by-freq function)

RETURN VALUE:

 the nearest sndfile object

SYNOPSIS:

```
(defmethod get-nearest (freq (sfp sndfile-palette) &optional ids random)
```

28.5.339   sndfile-palette/get-nearest-by-freq

[sndfile-palette] [Functions]

DATE:

 October 1st 2015, Edinburgh

DESCRIPTION:

 Given a frequency and a list of sndfile objects, return the sndfile from
 the list whose frequency is closest to the first argument.

ARGUMENTS:

 - a frequency in Hertz (number)
 - a simple list of sndfile objects, such as that contained in the data slot
  of a sndfile-palette group.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether a random file should be chosen when there are
   several sndfiles with the same frequency. (NB the function random-rep will
   be used so if you wanted repeatable results call (random-rep 1 t) in order
   to reset the random seed before calling clm-play or whatever context uses
   this function.) Default = NIL.

RETURN VALUE:

 the nearest sndfile object

SYNOPSIS:

(defun get-nearest-by-freq (freq sflist &optional random)

28.5.340   sndfile-palette/get-sndfiles

[sndfile-palette] [Functions]

DATE:

 5th September 2015, Edinburgh

DESCRIPTION:

Return a list of the full paths of sound files in the given path. Files
without the extensions aif, wav, aiff, and snd are ignored.

ARGUMENTS:

The full path to the folder where the sound files are, as a string

OPTIONAL ARGUMENTS:

- a list of folders to skip i.e. just the last folder name, not the
  complete path
- A single pattern (string) or list of patterns that the sound
  file name must have--just the filename, excluding path/folders and
  extension. If a list then all patterns must be in the file name, not
  just one of them. Default = NIL.
- Sim. to :insist except this/these are patterns none of which can
  be in the file name. Default = NIL.

RETURN VALUE:

A list of full paths as strings.

EXAMPLE:

```
(get-sndfiles "/music/hyperboles/snd/cello/samples/"
              '("short-percussive" "weird"))
-->
("/music/hyperboles/snd/cello/samples/1/g4-III-4-001.aif"
 "/music/hyperboles/snd/cello/samples/1/g4-III-4-002.aif"
 "/music/hyperboles/snd/cello/samples/1/g4-III-4-003.aif"
 "/music/hyperboles/snd/cello/samples/1/g4-III-4-004.aif"
 "/music/hyperboles/snd/cello/samples/10/cs5-I-5-9-13-4-001.aif"
 ...
```

SYNOPSIS:

```
(defun get-sndfiles (folder &optional skip insist resist)
```

28.5.341   sndfile-palette/kontakt-to-sfp

[sndfile-palette] [Functions]

DATE:

October 2nd 2015, Edinburgh

DESCRIPTION This function is only available on Mac OSX 64bit Intel machines
and requires the nki executable which since version 1.0.6 comes with
slippery-chicken (in the bin directory).

Using the public-domain C programme "nki tool" from the Linux Sampler
project (http://www.linuxsampler.org/nkitool/), convert a Kontakt .nki
sampler file to a sndfile-palette object. This will only include the sound
file and the MIDI note number it is mapped to, but it will mean that the
sndfile objects in the palette include the correct frequency. So using such
a sndfile-palette makes it possible to have clm-play act as a traditional
sampler, though of course much more is possible too.

The nki file could of course map more than one sample to any MIDI note
(e.g. for handling different velocities). By default all of these could be
included in the sndfile-palette, and the :snd-selector function passed to
clm-play could select the correct file based on velocity, for
instance. However, you may want to filter out some sound files. This is
where the :insist and :resist keyword arguments come in handy (see below).

NB If you get the error message "zpipe: invalid or incomplete deflate data"
then it's probably because the nki file is in Kontakt 4.2.2 (or higher)
format so can't be converted with this tool. You could try "save as" in
Kontakt 3 or 4 and rerunning this function, as those older programmes will
probably save in the older format (worked for me at least once).

ARGUMENTS:

- the path to the .nki file to process
- the path to the samples i.e. the sound files which the nki file uses

OPTIONAL ARGUMENTS:

Keyword arguments:
:insist
- :insist. A single pattern (string) or list of patterns that the sound
  file name must have--just the filename, excluding path/folders and
  extension. If a list then all patterns must be in the file name, not
  just one of them. Default = NIL i.e. accept all.
- :resist. Sim. to :insist except this/these are patterns none of which can
  be in the file name. Default = NIL.
- :group. The group ID that will be assigned to the sound files. Default =
  NIL whereby the name of the nki file will be used.
- :converter. The path to the nki tool executable. Default =
  /path/to/slippery-chicken/bin/nki

RETURN VALUE: EXAMPLE: SYNOPSIS:

```
#+(and mac-osx X86-64)
(defun kontakt-to-sfp (nki samples-path
                       &key insist resist group
                         (converter
                          (concatenate 'string
                                       cl-user::+slippery-chicken-home-dir+
                                       "bin/nki")))
```

28.5.342   sndfile-palette/make-sfp

[sndfile-palette] [Functions]

DESCRIPTION:

Make a sndfile-palette object. This object is a simple palette which checks
to make sure that all of the sound files in a given list exist for each
given ID.

Sound files are given as as single names, without the path and without the
extension. These can be given using the optional keyword arguments <paths>
and <extensions>.

NB Although this class is a palette and therefore a subclass of
recursive-assoc-list, the sound lists in this case cannot be nested beyond
a depth of two (as in example below).

ARGUMENTS:

- An ID for the palette.
- A list of lists that contains IDs for the names of one or more groups of
  sound files, each paired with a list of one or more names of existing
  sound files. The sound file names themselves can be paired with keywords
  from the sndfile class, such as :start, :end, and :frequency, to define
  and describe segments of a given sound file.

OPTIONAL ARGUMENTS:

keyword arguments:
- :paths. A list of one or more paths to where the sound files are located.
- :extensions. A list of one or more sound file extensions for the
  specified sound files. The default initialization for this slot of the
  sndfile-palette already contains ("wav" "aiff" "aif" "snd"), so this
  argument can often be left unspecified.

- :auto-freq. whether to do automatic frequency detection on the sound
  files. T would mean using a pitch detection routine, or a function can
  also be passed, whereupon it will be called with the path slot as
  argument (the idea is that the fundamental can be extracted from the file
  name). Default = NIL.
- :warn-not-found. T or NIL to indicate whether a warning should be printed
  to the Lisp listener if the specified sound file cannot be found.
  T = print warning. Default = T.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(let ((msfp (make-sfp 'sfp-test
                      '((sndfile-group-1
                         (test-sndfile-1))
                        (sndfile-group-2
                         (test-sndfile-2 test-sndfile-3
                          (test-sndfile-4 :frequency 261.61)))
                        (sndfile-group-3
                         ((test-sndfile-5 :start 0.006 :end 0.182)
                          test-sndfile-6)))
                      :paths '("/path/to/sound-files-dir-1/"
                               "/path/to/sound-files-dir-2/")))))
```

SYNOPSIS:

```
(defun make-sfp (id sfp &key paths (extensions '("wav" "aiff" "aif" "snd"))
                        auto-freq (warn-not-found t))
```

28.5.343   sndfile-palette/make-sfp-from-folder

[sndfile-palette] [Functions]

DATE:

 5th September 2015, Edinburgh

DESCRIPTION:

 Makes a sndfile-palette object from the sound files found in a specific
 folder (directory) on the file system. Allows any arbitrary levels of
 subfolders with the proviso that they'll be converted to a flat list, using

the subfolders as tags (e.g. dir/subdir1/subdir2 becomes
dir-subdir1-subdir2).

The folders can contain other files (they'll be ignored). Sound files are
those with extensions .aif .wav .aiff and .snd

ARGUMENTS:

 - the folder path, as a string.

OPTIONAL ARGUMENTS:

keyword arguments:
- :skip. a list of folders to skip i.e. just the last folder name, not the
  complete path. Default = NIL.
- :auto-freq. whether to do automatic frequency detection on the sound
  files. Default = NIL.
- :insist. A single pattern (string) or list of patterns that the sound
  file name must have--just the filename, excluding path/folders and
  extension. If a list then all patterns must be in the file name, not
  just one of them. Default = NIL.
- :resist. Sim. to :insist except this/these are patterns none of which can
  be in the file name. Default = NIL.

RETURN VALUE:

 a sndfile-palette object

SYNOPSIS: EXAMPLE:

```
(make-sfp-from-folder "/music/hyperboles/snd/cello/samples/"
                      :skip '("short-percussive" "weird"))
-->
SNDFILE-PALETTE: paths: NIL
                 extensions: (wav aiff aif snd)
                 num-snds: 92
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 16
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 16, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
```

```
NAMED-OBJECT: id: AUTO, tag: NIL,
data: (
LINKED-NAMED-OBJECT: previous: NIL, this: (1), next: (10)
NAMED-OBJECT: id: 1, tag: NIL,
data: (

SNDFILE-EXT: use: T, cue-num: 2, pitch: -1, pitch-curve: -1, bandwidth: -1,
             bandwidth-curve: -1, continuity: -1, continuity-curve: -1,
             weight: -1, weight-curve: -1, energy: -1, energy-curve: -1,
             harmonicity: -1, harmonicity-curve: -1, volume: -1,
             volume-curve: -1, loop-it: NIL, bitrate: 24, srate: 96000,
             num-frames: 1719753, bytes: 5159314, group-id: (1)
             followers: NIL

SNDFILE: path: /music/hyperboles/snd/cello/samples/1/g4-III-4-004.aif,
         snd-duration: 17.914093, channels: 1, frequency: 150.24414
         start: 0.0, end: 17.914093, amplitude: 1.0, duration 17.914093
         will-be-used: 0, has-been-used: 0
         data-consistent: T
...
|#
(defun make-sfp-from-folder (folder &key skip auto-freq insist resist
                                    (default-group 'default-group))
```

28.5.344   sndfile-palette/make-sfp-from-groups-in-wavelab-marker-file

[sndfile-palette] [Functions]

DESCRIPTION:

Automatically generate a sndfile-palette object using the specified sound
file from grouping defined in the specified wavelab marker file.

The <marker-file> argument can be passed as a list of marker files, in
which case these will first be concatenated.

ARGUMENTS:

- A string that is the name of the marker file, including the directory
  path and extension.
- A string that is the name of the sound file. This can either be a full
  directory path, file name, and extension, or just a base file name. If
  the latter, values for the optional arguments :paths and :extensions must
  also be specified.

OPTIONAL ARGUMENTS:

- :paths. NIL or a list of strings that are the directory paths to the
  specified sound files. If the sound file is passed with the directory
  path, this must be set to NIL. NB: The paths given here apply only to the
  sound files, not to the marker files. Default = NIL.
- :extensions. A list of strings that are the extensions to the given sound
  files. If the sound files are passed with their extensions, this must be
  set to NIL. Default = NIL.
- :warn-not-found. T or NIL to indicate whether to print a warning to the
  listener if the specified sound file is not found. T = print a
  warning. Default = NIL.
- :sampling-rate. An integer that is the sampling rate of the specified
  sound file. Changing this value will alter the start-times determined for
  each sound segment. Default = 44100.
- :print. T or NIL to indicate whether feedback about the groups found and
  created should be printed to the listener. T = print. Default = T.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
(make-sfp-from-groups-in-wavelab-marker-file
 "/path/to/24-7.mrk"
 "24-7"
 :paths '("/path/to/sndfile/directory/")
 :sampling-rate 44100
 :extensions '("wav"))

=>
24 markers read from /path/to/24-7.mrk
Adding tapping: 2.753 -> 4.827
Adding tapping: 5.097 -> 6.581
Adding tapping: 6.763 -> 8.538
Adding splinter: 13.878 -> 15.993
Adding tapping: 16.338 -> 18.261
Adding splinter: 19.403 -> 25.655

tapping: 4 sounds
splinter: 2 sounds
```

SYNOPSIS:

```
(defun make-sfp-from-groups-in-wavelab-marker-file (marker-file sndfile
                                                     &key
```

```
                                                     paths
                                                     extensions
                                                     warn-not-found
                                                     (sampling-rate 44100)
                                                     (print t))
```

28.5.345   sndfile-palette/make-sfp-from-wavelab-marker-file

[sndfile-palette] [Functions]

DESCRIPTION:

Automatically create a sndfile-palette object from the specified wavelab
marker file and the specified sound file (from which the marker file must
have been generated).

The function will produce a sndfile-palette object with multiple groups,
each of which consists of the number of sound file segments specified using
the :snds-per-group argument (defaults to 8). By default the segments will
be collected into the groups in chronological order. If the optional
:random-every argument is given a value, every nth group will consist of
random segments instead.

The sound file segments of each group will correspond to the time points
stored in the marker file.

The <marker-file> argument can consist of a list of marker files, in which
case these would first be concatenated.

NB: Be aware that marker files created on operating systems differing from
    the one on which this function is called might trigger errors due to
    newline character mismatches.

ARGUMENTS:

- A string that is the name of the marker file, including the directory
  path and extension.
- A string that is the name of the sound file. This can either be a full
  directory path, file name, and extension, or just a base file name. If
  the latter, values for the optional arguments :paths and :extensions must
  also be specified.

OPTIONAL ARGUMENTS:

keyword arguments:

  - :snds-per-group. An integer that is the number of sound file segments to
    include in each group. Default = 8.
  - :random-every. An integer to indicate that every nth group is to consist
    of random (rather than chronologically consecutive) sound file segments.
    Default = 999999 (i.e. essentially never)
  - :paths. NIL or a list of strings that are the directory paths to the
    specified sound files. If the sound file is passed with the directory
    path, this must be set to NIL. NB: The paths given here apply only to the
    sound files, not to the marker files. Default = NIL.
  - :sampling-rate. An integer that is the sampling rate of the specified
    sound file. Changing this value will alter the start-times determined for
    each sound segment. Default = 44100.
  - :extensions. A list of strings that are the extensions to the given sound
    files. If the sound files are passed with their extensions, this must be
    set to NIL. Default = NIL.
  - :warn-not-found. T or NIL to indicate whether to print a warning to the
    listener if the specified sound file is not found. T = print a
    warning. Default = NIL.
  - :name. The name for the overall sndfile-palette and the base name for
     each group within (these will have a suffix that is an auto-incrementing
     number e.g. 'auto would become 'auto1 'auto2 etc.). Default = 'auto.

RETURN VALUE:

 A sndfile-palette object.

EXAMPLE:

```
(make-sfp-from-wavelab-marker-file
  "/path/to/24-7.mrk"
 "24-7"
 :snds-per-group 2
 :random-every 3
 :paths '("/path/to/sound-file/directory/")
 :sampling-rate 44100
 :extensions '("wav"))

=>
SNDFILE-PALETTE: paths: (/Volumes/JIMMY/SlipperyChicken/sc/test-suite/)
                extensions: (wav)
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                     num-data: 8
                     linked: NIL
                     full-ref: NIL
ASSOC-LIST: warn-not-found NIL
```

```
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 8, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: AUTO, tag: NIL,
data: (
NAMED-OBJECT: id: "auto1", tag: NIL,
data: (

SNDFILE: path: /Volumes/JIMMY/SlipperyChicken/sc/test-suite/24-7.wav,
        snd-duration: 29.652811, channels: 2, frequency: 261.62555
        start: 0.09142857, end: 1.0361905, amplitude: 1.0, duration 0.94476193
        will-be-used: 0, has-been-used: 0
        data-consistent: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "24-7", tag: NIL,
data: /Volumes/JIMMY/SlipperyChicken/sc/test-suite/24-7.wav
[...]
```

SYNOPSIS:

```
(defun make-sfp-from-wavelab-marker-file (marker-file sndfile
                                          &key
                                          (snds-per-group 8)
                                          (random-every 999999) ;; i.e. never
                                          paths
                                          (sampling-rate 44100)
                                          extensions
                                          ;; MDE Fri Oct  5 14:04:08 2012
                                          (name 'auto)
                                          warn-not-found)
```

28.5.346   sndfile-palette/set-frequency-from-filename

[sndfile-palette] [Methods]

DATE:

 October 1st 2015, Edinburgh

DESCRIPTION:

 Set the frequency slot of every sound file object in the palate to a value
 extracted from the file name. This is intended but of course not restricted
 to palettes created from sample libraries, where, as is often the case, the
 file name of each sample contains the pitch of the file. For example, from

the Akoustik Piano sample library we have the file
Stein-R(A1)-V(220)-dB(2446)-T(MF)-M(123)-P(404-03-01).wav. In this case the
A1 in the first parentheses indicate the pitch. By writing and passing a
short parsing function to read the pitch out of such a file name we can
easily process each sound file in the palette. See akoustik-piano-name
below for an example of how to process such file names.

See also make-sfp-from-folder.

ARGUMENTS:

 - The sound file palette object

OPTIONAL ARGUMENTS:

Keyword arguments:
 - :groups. A single group or list of groups (IDs) for the sound files we
   wish to process.
 - :name-fun.The function for processing a single name. Of course the names
   must be consistent and each sound file must be able to be processed by
   this single function. NB for reasons of similar usage in get-spectra-al
   (see spectra.lsp) this function actually must return the MIDI note number
   (may be a floating point for microtonal applications), which is then
   converted by the sndfile class to a frequency in Hertz.
 - :on-error. The function to be called when the name function cannot
   determine the frequency of the sound file. This could be #'error (the
   default), #'warn, or nil if nothing is to be done on failure.

RETURN VALUE:

 The sndfile-palette object after processing.

SYNOPSIS:

```
(defmethod set-frequency-from-filename
    ((sfp sndfile-palette)
     &key groups (name-fun #'akoustik-piano-name) (on-error #'error))
```

28.5.347   palette/sndfilenet

[palette] [Classes]

NAME:

 sndfile-palette

```
File:              sndfilenet.lsp

Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list -> recursive-assoc-list ->
                   palette -> sndfile-palette -> sndfilenet

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           Implementation of the sndfilenet class which extends
                   sndfile-palette to add functionality for loading and
                   playing sounds within MaxMSP.

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     23rd October 2017, Essen

$$ Last modified:  14:02:11 Mon Apr 13 2020 CEST

SVN ID: $Id$
```

28.5.348   sndfilenet/(setf next-sfe)

[sndfilenet] [Methods]

DATE:

```
October 21st 2017, Essen
```

DESCRIPTION:

```
A setf :after method. Set the next-sfe slot to be a sndfile-ext object. If
it is already, leave it as is. If it's an integer, assume it's a cue number
and get the right object, otherwise get the object with this ID.

If this is causing problems, use (setf (slot-value sfn 'next-sfe) ... )

If we're trying to set using a reference/id like '(group1 snd3) then it
will have to be a two-element list.
```

SYNOPSIS:

```
(defmethod (setf next-sfe) :after (value (sfn sndfilenet))
```

28.5.349   sndfilenet/analyse-followers

[sndfilenet] [Methods]

DESCRIPTION:

```
Using the followers slots of each sndfile in the palette, go through each
sndfile in the palette and generate a large number of following sounds,
i.e. emulate max-play. The results of the follow-on process are then
analysed and a warning will be issued if any sndfile seems to dominate
(defined as being present at least twice as many times as its 'fair share',
where 'fair share' would mean an even spread for all the sound files in the
palette).
```

ARGUMENTS:

```
- The sndfilenet object.
```

OPTIONAL ARGUMENTS:

```
- integer: How many times to repeat the generation process.  Default = 1000.
- T or NIL for verbose warnings: if T, the emulation list of sound files
  will be printed. Default NIL.
```

RETURN VALUE:

```
T or NIL depending on whether the analysis detects an even spread or not.
```

SYNOPSIS:

```
(defmethod analyse-followers ((sfn sndfilenet)
                              &optional (depth 1000) verbose)
```

28.5.350   sndfilenet/auto-cue-nums

[sndfilenet] [Methods]

DESCRIPTION:

```
Set the cue-num slot of every sndfile-ext object in the palette to be an
ascending integer starting at 2. NB If a sndfile has it's :use slot set to
NIL it won't be given a cue number.
```

ARGUMENTS:

```
- a sndfilenet object.
```

RETURN VALUE:

```
The cue number of the last sndfile-ext object.
```

SYNOPSIS:

```
(defmethod auto-cue-nums ((sfn sndfilenet))
```

28.5.351   sndfilenet/auto-followers

[sndfilenet] [Methods]

DATE:

```
October 20th 2017
```

DESCRIPTION:

```
Automatically generate follower sound files for every sound file in the
palette. These are a mixture of the other files in the same group plus one
from another group, so that we'll move through.
```

ARGUMENTS:

```
- a sndfilenet object
```

RETURN VALUE:

```
T
```

SYNOPSIS:

```
(defmethod auto-followers ((sfn sndfilenet))
```

28.5.352   sndfilenet/get-snd-with-cue-num

[sndfilenet] [Methods]

DATE:

```
December 21st 2012
```

DESCRIPTION:

Return the (first, but generally unique) sndfile object which has the
given cue-num slot.

ARGUMENTS:

- the sndfilenet object.
- the cue number (integer).

RETURN VALUE:

The sndfile/sndfile-ext object with the given cue number or NIL if it can't
be found.

SYNOPSIS:

(defmethod get-snd-with-cue-num ((sfn sndfilenet) cue-num)

28.5.353 sndfilenet/make-sfn

[sndfilenet] [Functions]

DESCRIPTION:

Make a sndfilenet object. This object is a simple palette which checks
to make sure that all of the sound files in a given list exist for each
given ID.

Sound files are given as as single names, without the path and without the
extension. These can be given using the optional keyword arguments <paths>
and <extensions>.

NB Although this class is a palette and therefore a subclass of
recursive-assoc-list, the sound lists in this case cannot be nested beyond
a depth of two (as in example below).

ARGUMENTS:

- An ID for the palette.
- A list of lists that contains IDs for the names of one or more groups of
  sound files, each paired with a list of one or more names of existing
  sound files. The sound file names themselves can be paired with keywords
  from the sndfile class, such as :start, :end, and :frequency, to define
  and describe segments of a given sound file.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :paths. A list of one or more paths to where the sound files are located.
- :extensions. A list of one or more sound file extensions for the
  specified sound files. The default initialization for this slot of the
  sndfilenet already contains ("wav" "aiff" "aif" "snd"), so this
  argument can often be left unspecified.
- :warn-not-found. T or NIL to indicate whether a warning should be printed
  to the Lisp listener if the specified sound file cannot be found.
  T = print warning. Default = T.
```

RETURN VALUE:

```
Returns NIL.
```

EXAMPLE:

```
;;; NB won't work unless you change /path/to/ below
(let ((msfn (make-sfn 'sfn-test
                      '((sndfile-group-1
                         (test-sndfile-1))
                        (sndfile-group-2
                         (test-sndfile-2 test-sndfile-3
                          (test-sndfile-4 :frequency 261.61)))
                        (sndfile-group-3
                         ((test-sndfile-5 :start 0.006 :end 0.182)
                          test-sndfile-6)))
                      :paths '("/path/to/sound-files-dir-1/"
                               "/path/to/sound-files-dir-2/")))))
```

SYNOPSIS:

```
(defun make-sfn (id sfn &key paths (extensions '("wav" "aiff" "aif" "snd"))
                        auto-freq with-followers (warn-not-found t))
```

28.5.354   sndfilenet/max-play

[sndfilenet] [Methods]

DESCRIPTION:

```
This generates the data necessary to play the next sound in the current
sound's followers list. See the sndfile-ext method for details.
```

ARGUMENTS:

- The sndfilenet object.
- The fade (in/out) duration in seconds.
- The maximum loop duration in seconds.
- The time to trigger the next file, as a percentage of the current
  sndfile-ext's duration.

OPTIONAL ARGUMENTS:

- whether to print data to the listener as it is generated. Default = NIL.

RETURN VALUE:

 A list of values returned by the sndfile-ext method.

SYNOPSIS:

```
(defmethod max-play ((sfn sndfilenet) fade-dur max-loop start-next
                     &optional print)
```

28.5.355   sndfilenet/osc-send-cue-nums

[sndfilenet] [Methods]

DESCRIPTION:

 Send via OSC the cue number of each sound file in a form that a Max sflist~
 can process and store (which will include the path and start/stop data: see
 sndfile-ext::max-cue for details).

 E.g. preload 231 triplet-mphonic-001.wav 0. 3300.612305 0 1.

 Note that the individual preload data is sent by this method explicitly over
 the UDP network. This means you'll need a [udpreceive] somewhere in the max
 patch to catch these and that [osc-sc-eval] won't send these out of its
 outlet, rather the number of cues will be output there. Also note that the
 usual IDs prepended to results from osc-eval are not included in the preload
 sent over UDP so a separate parser looking for this data will be needed,
 e.g.

 [udpreceive 8091][fromsymbol][route preload][prepend preload][sflist~ mysfl]

ARGUMENTS:

- the sndfilenet object.

RETURN VALUE:

The number of cue numbers sent. NB This is not the same as the last cue
number as cues start from 2.

SYNOPSIS:

```
#+(and darwin sbcl)
(defmethod osc-send-cue-nums ((sfn sndfilenet))
```

28.5.356   sndfilenet/reset

[sndfilenet] [Methods]

DESCRIPTION:

Reset the followers' slot circular list to the beginning or to <where>

ARGUMENTS:

 - the sndfilenet object.

OPTIONAL ARGUMENTS:

 - an integer to set the point at which to restart. This can be higher than
   the number of followers as it will wrap. Default = nil (which equates to
   0 lower down in the class hierarchy).
 - whether to issue a warning if <where> is greater than the number of
   followers (i.e. that wrapping will occur). Default = T.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod reset ((sfn sndfilenet) &optional where (warn t))
```

28.5.357   recursive-assoc-list/parcel-data

[recursive-assoc-list] [Methods]

DATE:

10 Apr 2010

DESCRIPTION:

 Put all the data of a given recursive-assoc-list object into a new
 named-object at the top level of that recursive-assoc-list object; i.e. add
 a level of recursion. This is a means of making a collection of data
 before perhaps adding more with potentially conflicting ids.

ARGUMENTS:

 - A recursive-assoc-list object.
 - A symbol that is new the top-level id for the current data

RETURN VALUE:

 The new recursive-assoc-list object.

EXAMPLE:

```
;; Collect all the data contained within the object 'mixed-bag and store it at
;; the top-level of 'mixed-bag within a new named-object with the id 'potpourri
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (parcel-data ral 'potpourri))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 1, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: FROM-PARCEL-DATA,
data: (
NAMED-OBJECT: id: POTPOURRI, tag: NIL,
```

```
data:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: NIL,
data: (
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM
[...]
```

SYNOPSIS:

```
(defmethod parcel-data ((ral recursive-assoc-list) new-id)
```

28.5.358   recursive-assoc-list/r-count-elements

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Return the total number of elements recursively (across all depths) of the
 given recursive-assoc-list object.

ARGUMENTS:

 - A recursive-assoc-list object.

RETURN VALUE:

 An integer.

EXAMPLE:

```
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
```

```
                                                     (fox hole)))
                                          (white ribbon)))))))))))
  (r-count-elements ral))
```

=> 8

SYNOPSIS:

```
(defmethod r-count-elements ((ral recursive-assoc-list))
```

28.5.359   recursive-assoc-list/ral-econs

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Automatically create new recursive-assoc-list objects.

 This method assumes that any existing key that may be referenced will be
 associated with data that is already a list, to the end of which the new
 data will be added (an error will be signalled if this is not the case.)

ARGUMENTS:

 - The data which is to be added.
 - The key to which the data is to be added (see above note for cases where
   this key already exists).
 - The recursive-assoc-list object to which this data is to be added.

RETURN VALUE:

 The new data added.

EXAMPLE:

```
;;; Make an empty recursive-assoc-list object and add key/data pairs to the top
;;; level.
(let ((ral (make-ral nil nil)))
  (print (get-all-refs ral))
  (ral-econs 'beam 'jim ral)
  (ral-econs 'turkey 'wild ral)
  (ral-econs 'roses 'four ral)
  (print (get-all-refs ral))
  (print (get-data-data 'wild ral)))
```

```
=>
NIL
((JIM) (WILD) (FOUR))
(TURKEY)

;;; Add data to existing keys within a given recursive-assoc-list object
;;; Note that the data VELVET must be a list for this to succeed

(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue (velvet))
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (print (get-all-refs ral))
  (ral-econs 'underground '(four violets blue) ral)
  (print (get-data-data '(four violets blue) ral)))

=>
((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
 (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE))
(VELVET UNDERGROUND)
```

SYNOPSIS:

```
(defmethod ral-econs (data key (ral recursive-assoc-list))
```

28.5.360 recursive-assoc-list/recursivep

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Check whether the data in a recursive-assoc-list object is really
 recursive.

ARGUMENTS:

 - A recursive-assoc-list object.

RETURN VALUE:

```
 T or NIL to indicate whether or not the tested data is recursive.
 T = recursive.
```

EXAMPLE:

```
;; The data in this recursive-assoc-list object is really recursive, and
;; the method therefore returns T
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (recursivep ral))

=> T

;; The data in this recursive-assoc-list object is not actually recursive, and
;; the method therefore returns NIL
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four roses)))))
  (recursivep ral))

=> NIL
```

SYNOPSIS:

```
(defmethod recursivep ((ral recursive-assoc-list))
```

28.5.361   recursive-assoc-list/relink-named-objects

[recursive-assoc-list] [Methods]

DESCRIPTION:

```
 This method is essentially the same as the method link-named-objects, but
 resets the LINKED slot to NIL and forces the link-named-objects method to
 be applied again.
```

ARGUMENTS:

- A recursive-alloc-list object.

RETURN VALUE:

A recursive-alloc-list object.

EXAMPLE:

```
;; Usage as presented here; see the documentation for method link-named-objects
;; for more detail
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (relink-named-objects ral))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 8
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: MIXED-BAG, tag: NIL,
data: (
LINKED-NAMED-OBJECT: previous: NIL, this: (JIM), next: (WILD)
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM
[...]
```

SYNOPSIS:

```
(defmethod relink-named-objects ((ral recursive-assoc-list))
```

28.5.362   recursive-assoc-list/remove-data

[recursive-assoc-list] [Methods]

DATE:

February 1st 2016, Edinburgh

DESCRIPTION:

Convenience function to allow objects to be removed from a
recursive-assoc-list (and subclasses e.g. palettes and maps)

ARGUMENTS:

- the recursive assoc-list object
- as many keys as you like. If none are passed then all are removed.

RETURN VALUE:

The recursive-assoc-list object's new data list, i.e. with elements
removed. Could of course be NIL if you've not passed any keys.

SYNOPSIS:

(defmethod remove-data ((ral recursive-assoc-list) &rest keys)

28.5.363   recursive-assoc-list/rmap

[recursive-assoc-list] [Methods]

DESCRIPTION:

Recurse over the objects in a recursive-assoc-list and call the given
function for each each named-object. See also assoc-list's map-data method
which does pretty much the same but acting on each named-object's data
rather than the named-object itself.

ARGUMENTS:

- the recursive-assoc-list object
- the function to call (function object)

OPTIONAL ARGUMENTS:

- &rest further arguments to be passed to the function after the
  named-object from the recursive-assoc-list.

RETURN VALUE:

The recursive-assoc-list object

EXAMPLE:

```
(let ((ral (make-ral 'mixed-bag
                      '((jim beam)
                        (wild turkey)
                        (four ((roses red)
                               (violets ((blue velvet)
                                         (red ((dragon den)
                                               (viper nest)
                                               (fox hole)))
                                         (white ribbon)))))))))
  (rmap ral #'print))
=>
NAMED-OBJECT: id: JIM, tag: NIL,
data: BEAM
**************
NAMED-OBJECT: id: WILD, tag: NIL,
data: TURKEY
**************
NAMED-OBJECT: id: ROSES, tag: NIL,
data: RED
**************
NAMED-OBJECT: id: BLUE, tag: NIL,
data: VELVET
**************
NAMED-OBJECT: id: DRAGON, tag: NIL,
data: DEN
**************
NAMED-OBJECT: id: VIPER, tag: NIL,
data: NEST
**************
NAMED-OBJECT: id: FOX, tag: NIL,
data: HOLE
**************
NAMED-OBJECT: id: WHITE, tag: NIL,
data: RIBBON
**************
T
```

SYNOPSIS:

```
(defmethod rmap ((ral recursive-assoc-list) function &rest arguments)
```

28.5.364   recursive-assoc-list/sc-map

[recursive-assoc-list] [Classes]

NAME:

```
sc-map
```

```
File:           sc-map.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist ->
                circular-sclist -> assoc-list -> recursive-assoc-list ->
                sc-map

Version:        1.0.11

Project:        slippery chicken (algorithmic composition)

Purpose:        Implementation of the sc-map class for mapping rhythmic
                sequences, chords etc. to specific parts of a piece.
                The extension to the recursive-assoc-list class is in the
                data returned when get-data-from-palette is called: being
                a map, the data returned by the superclass get-data
                function is actually a reference into a palette.  Instead
                of just returning this reference, with
                get-data-from-palette we then use this as a lookup into
                the palette slot.  If the reference happens to be a list,
                then each element of the list is used as a reference into
                the palette and the resulting objects are returned in a
                list.

                TODO: When in a list of references, perhaps the rthm-seq
                references for a section, a single reference is also a
                list this can be one of two things: the reference is to a
                recursive palette, whereupon the data will simply be
                returned for that reference; or, the reference is a list
                of references that together build up an object consisting
                of the referenced smaller objects.  This is the case
                when, for example, 4-bar sequences in one or more
                instruments are accompanied by groups of 4 single bar
                sequences in others:

                (2
                 ((bsn ((r1-1 r1-2 r1-3 r1-5) 20 1 ...))
                  (trb (2 23 3 ...)))))
```

```
Author:           Michael Edwards: m@michael-edwards.org

Creation date:    March 21st 2001

$$ Last modified:  17:18:41 Fri Sep 18 2020 CEST

SVN ID: $Id$
```

28.5.365   sc-map/change-map

[sc-map] [Classes]

NAME:

```
change-map

File:             change-map.lsp

Class Hierarchy:  named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list ->
                  sc-map -> change-map

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Implementation of the change-map class where, unlike
                  normal sc-maps (data is given for each sequence) gives
                  data sporadically when the parameter changes, for
                  instance tempo.

                  It is assumed that maps will be typed in the order in
                  which sections occur so that previous-data slots can be
                  kept up-to-date; also, unless all data will be given,
                  that the sections (but not instrument ids) will be in
                  integer sequential order so that nearest sections can be
                  returned when given a non-existent section reference.

                  For example, the following change-map indicates tempo
                  (though tempo-maps have their own class now).
                  It has sections within sections so that the tempo for
                  section '(1 2 3) can be defined, that is, section 1 has
                  subsections 1, 2, and 3 and subsection 2 has further
                  subsections 1, 2, and 3.  This can be nested to any
                  depth.  The tempo information itself is given in sublists
                  where (3 27) means that in the third sequence of the
```

section, the tempo is 27. (3 2 27) means the 2nd bar of
the third sequence has tempo 27: when only two numbers
are in the list, bar 1 is assumed. The trick is, that
this tempo then remains, as would be expected, until the
next tempo change is indicated, which means that
requesting the tempo of section (2 2 3) with any sequence
and bar in the map below would return 25, because that is
the last tempo given in section 1 and no tempo is defined
for section 2.

```
(setf x
  (make-change-map
   'test nil
   '((0 ((3 27) (9 3 45)))
     (1
      ((1 ((1 21) (5 28) (8 35) (3 2 40) (3 1 54)))
       (2
        ((1 ((1 23) (6 28) (18 35)))
         (2 ((2 2 24) (7 28) (18 22)))
         (3 ((3 34) (7 28) (18 42)))))
       (3 ((1 22) (5 34) (10 5 25)))))
     (4
      ((1 ((1 21) (5 28) (8 36) (3 2 40) (3 1 55)))
       (2 ((1 22) (5 34) (10 5 103)))))
     (5 ((2 28) (6 3 45)))
     (10
      ((1 ((1 21) (5 28) (8 37) (3 2 40) (3 1 56)))
       (2 ((1 22) (5 34) (10 5 27)))))))))
```

You have to be careful with change-maps however as the
nesting is flexible and therefore ambiguous. For
instance, in the following the bcl, tape1 etc. ids are
not subsections of section 1, rather these are the hint
pitches assigned to the instruments in section 1 (which
has no subsections). This is where the last-ref-required
class slot comes in: If this slot is t (this is the
second argument to make-change-map) then the last
reference in a call to cm-get-data is always respected,
i.e. not the last data given will be returned when the
section doesn't exist, rather the last data for this
reference. E.g. In the following map, if
last-ref-required were nil, then the call to
(cm-get-data x '(2 tape2) 1) would fail (because we can't
find nearest data when references aren't numbers), but
because it's t, we get the last data given for tape2 and
return cs5.

```
                       (setf x
                         (make-change-map
                            'hint-pitches t
                            '((1 ((bcl ((1 a4) (2 b4) (3 c5) (4 d6)))
                                  (tape1 ((1 a3) (2 ds2) (3 e4)))
                                  (tape2 ((1 a3) (2 ds2) (3 cs5)))
                                  (tape3 ((1 a3) (2 ds2) (3 eqf4)))))
                               (2 ((bcl ((1 a4) (2 b4) (3 c5) (4 d6)))
                                  (tape1 ((1 a3) (2 ds2) (5 fs4)))))
                               (3 ((bcl ((1 a4) (2 b4) (3 c5) (4 d6)))
                                  (tape1 ((1 a3) (2 ds2) (5 f4)))))))))
```

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 2nd April 2001

 $$ Last modified: 17:34:14 Tue Jan 14 2020 CET

 SVN ID: Id

28.5.366   change-map/cm-get-data

[change-map] [Methods]

DESCRIPTION:

 Return the data for a specified section and player of a given change-map
 object.

 NB: The <section> argument may require the ID of the player as well if
     used, for example, with the instrument-change-map subclass of this
     class .

ARGUMENTS:

 - A change-map object.
 - A simple key reference into the given change-map object or a list of
   references. NB: This reference may require a player ID if, for example,
   used with an instrument-change-map subclass of this class.

OPTIONAL ARGUMENTS:

 - The ID of the sequence from which to return the change-map data.
 - An integer that is the bar number for which to return the change-map
   data.

RETURN VALUE:

 The change-map data stored at the specified location within the given
 change-map.

EXAMPLE:

```
;;; An example using the instrument-change-map subclass of change-map

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                 (3 ((sax ((3 alto-sax) (4 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))))
                        (3 ((sax (1 1 1 1 1)))))))))
  (cm-get-data (instrument-change-map mini) '(2 sax) 4))

=> ALTO-SAX
```

SYNOPSIS:

```
(defmethod cm-get-data ((cm change-map) section
                        &optional (sequence 1) (bar 1))
```

28.5.367   change-map/find-nearest

[change-map] [Methods]

DESCRIPTION:

 Return the nearest change-data object to the specified section within the
 given change-map. NB: The section may require a player ID if used, for
 example, with an instrument-change-map subclass of change-map.

ARGUMENTS:

  - A section indication, either as a single reference ID or a list of
    reference IDs into the given change-map.
  - A change-map object.

RETURN VALUE:

 Returns a change-data object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                 (3 ((sax ((3 alto-sax) (4 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))))
                        (3 ((sax (1 1 1 1 1)))))))))
  (find-nearest '(4 sax) (instrument-change-map mini)))

=>
CHANGE-DATA:
           previous-data: TENOR-SAX,
           last-data: TENOR-SAX
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: (2 SAX), this: (3 SAX), next: NIL
NAMED-OBJECT: id: SAX, tag: NIL,
data: ((3 1 ALTO-SAX) (4 1 TENOR-SAX))
```

SYNOPSIS:

```
(defmethod find-nearest (section (cm change-map))
```

28.5.368   change-map/instrument-change-map

[change-map] [Classes]

NAME:

```
instrument-change-map
```

```
File:              instrument-change-map
```

```
Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list -> recursive-assoc-list ->
                   sc-map -> change-map -> instrument-change-map
```

```
Version:           1.0.11
```

```
Project:           slippery chicken (algorithmic composition)
```

```
Purpose:           Extends change-map to check that instruments defined in
                   the map have data for the first bar of the first section.
```

```
Author:            Michael Edwards: m@michael-edwards.org
```

```
Creation date:     12th April 2002
```

```
$$ Last modified:  13:40:34 Fri Mar 17 2017 GMT
```

```
SVN ID: $Id$
```

28.5.369   instrument-change-map/get-first-for-player

[instrument-change-map] [Methods]

DESCRIPTION:

Return the first instrument object assigned to a given player in cases
where a player has been assigned more than one instrument.

ARGUMENTS:

- An instrument-change-map
- The ID of the player for whom to return the first instrument.

RETURN VALUE:

The ID of the first instrument object assigned to the specified player.

EXAMPLE:

```
(let ((mini
      (make-slippery-chicken
```

```
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                    (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1)))))))))
  (get-first-for-player (instrument-change-map mini) 'sax))

=> ALTO-SAX, T
```

SYNOPSIS:

```
(defmethod get-first-for-player ((icm instrument-change-map)
                                 player)
```

28.5.370   instrument-change-map/make-instrument-change-map

[instrument-change-map] [Functions]

DESCRIPTION:

 Create an instrument-change-map object.

ARGUMENTS:

 - An ID for the instrument-change-map to be created.
 - A list of lists. The top level of these lists consists of the section IDs
   for the given slippery-chicken object paired with lists of data for the
   specified players for each section. Each player list then consists of the
   ID of the player paired with a list of two-item lists pairing measure
   numbers with the instrument to which that player is to change, e.g.:

```
   '((1 ((fl ((1 flute) (3 piccolo) (5 flute)))
         (cl ((1 b-flat-clarinet) (2 bass-clarinet) (6 b-flat-clarinet)))))
      (2 ((fl ((2 piccolo) (4 flute)))
         (cl ((2 bass-clarinet) (3 b-flat-clarinet))))))
```

OPTIONAL ARGUMENTS:

 - :warn-not-found. T or NIL to indicate whether a warning is printed when
   an index which doesn't exist is used for look-up. T = warn. Default = T.

RETURN VALUE: EXAMPLE:

```
(make-instrument-change-map
 'icm-test
 '((1 ((fl ((1 flute) (3 piccolo) (5 flute)))
       (cl ((1 b-flat-clarinet) (2 bass-clarinet) (6 b-flat-clarinet)))))
   (2 ((fl ((2 piccolo) (4 flute)))
       (cl ((2 bass-clarinet) (3 b-flat-clarinet)))))))

=>

INSTRUMENT-CHANGE-MAP:
CHANGE-MAP: last-ref-required: T
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: NIL
                      num-data: 4
                      linked: T
                      full-ref: NIL
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: ICM-TEST, tag: NIL,
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data:
CHANGE-MAP: last-ref-required: T
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: NIL
                      num-data: 2
                      linked: T
                      full-ref: (1)
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "sub-ral-of-ICM-TEST", tag: NIL,
data: (
CHANGE-DATA:
           previous-data: NIL,
           last-data: FLUTE
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: (1 FL), next: (1 CL)
NAMED-OBJECT: id: FL, tag: NIL,
data: ((1 1 FLUTE) (3 1 PICCOLO) (5 1 FLUTE))
**************
```

```
CHANGE-DATA:
            previous-data: NIL,
            last-data: B-FLAT-CLARINET
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: (1 FL), this: (1 CL), next: (2 FL)
NAMED-OBJECT: id: CL, tag: NIL,
data: ((1 1 B-FLAT-CLARINET) (2 1 BASS-CLARINET) (6 1 B-FLAT-CLARINET))
**************
)
**************

**************


NAMED-OBJECT: id: 2, tag: NIL,
data:
CHANGE-MAP: last-ref-required: T
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: NIL
                      num-data: 2
                      linked: T
                      full-ref: (2)
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "sub-ral-of-ICM-TEST", tag: NIL,
data: (
CHANGE-DATA:
            previous-data: FLUTE,
            last-data: FLUTE
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: (1 CL), this: (2 FL), next: (2 CL)
NAMED-OBJECT: id: FL, tag: NIL,
data: ((2 1 PICCOLO) (4 1 FLUTE))
**************


CHANGE-DATA:
            previous-data: B-FLAT-CLARINET,
            last-data: B-FLAT-CLARINET
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: (2 FL), this: (2 CL), next: NIL
NAMED-OBJECT: id: CL, tag: NIL,
```

```
data: ((2 1 BASS-CLARINET) (3 1 B-FLAT-CLARINET))
**************
)
**************

**************
)
```

SYNOPSIS:

```
(defun make-instrument-change-map (id icm &key (warn-not-found nil))
```

28.5.371   change-map/simple-change-map

[change-map] [Classes]

NAME:

```
 simple-change-map

 File:              simple-change-map.lsp

 Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                    circular-sclist -> assoc-list -> recursive-assoc-list ->
                    sc-map -> change-map -> simple-change-map

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           Implementation of the simple-change-map class which gives
                    data associated with a non-recursive list of number ids.
                    For example, good for specifying data which changes at
                    specific bar numbers.

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     March 31st 2002

 $$ Last modified: 20:16:16 Mon May 14 2012 BST

 SVN ID: $Id$
```

28.5.372   simple-change-map/make-simple-change-map

[simple-change-map] [Functions]

DESCRIPTION:

 Create a simple-change-map object, which stores data associated with a
 non-recursive list of number IDs. This object could be good, for example,
 for specifying data which changes at specific bar numbers.

ARGUMENTS:

 - An ID for the simple-change-map object to be created.
 - A list of non-recursive lists consisting of ID/data pairs, of which the
   first item is a numerical ID.

RETURN VALUE:

 A simple-change-map object.

EXAMPLE:

```
(make-simple-change-map 'bar-map '((1 3) (34 3) (38 4)))

=>
SIMPLE-CHANGE-MAP:
CHANGE-MAP: last-ref-required: NIL
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: NIL
                      num-data: 3
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: BAR-MAP, tag: NIL,
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data: 3
**************


NAMED-OBJECT: id: 34, tag: NIL,
data: 3
**************
```

```
NAMED-OBJECT: id: 38, tag: NIL,
data: 4
**************
)
**************
```

SYNOPSIS:

```
(defun make-simple-change-map (id scm)
```

28.5.373   simple-change-map/scm-get-data

[simple-change-map] [Methods]

DESCRIPTION:

 Get the data associated with the specified key within a given
 simple-change-map object.

ARGUMENTS:

 - An integer that is an existing key ID within the given simple-change-map
   object.
 - A simple-change-map-object.

RETURN VALUE:

 The data associated with the specified key ID, as a named object.

EXAMPLE:

```
(let ((scm (make-simple-change-map 'bar-map '((1 3) (34 3) (38 4)))))
  (scm-get-data 34 scm))

=>
NAMED-OBJECT: id: 34, tag: NIL,
data: 3
```

SYNOPSIS:

```
(defmethod scm-get-data (ref (scm simple-change-map))
```

28.5.374   simple-change-map/tempo-map

[simple-change-map] [Classes]

NAME:

```
tempo-map

 File:             tempo-map.lsp

 Class Hierarchy:  named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list -> recursive-assoc-list ->
                   sc-map -> change-map -> simple-change-map -> tempo-map

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Allow the specification of tempi by just a number
                   (defaulting to crotchet number) or a list where the first
                   element would be the beat, the second the speed.

 Author:           Michael Edwards: m@michael-edwards.org

 Creation date:    6th May 2006

 $$ Last modified: 15:49:41 Fri Oct 11 2013 BST

 SVN ID: $Id$
```

28.5.375   sc-map/count-ref

[sc-map] [Methods]

DATE:

```
February 13th 2018, Heidhausen
```

DESCRIPTION:

```
Count the number of times a map reference occurs, e.g. how often a
particular rthm-seq or set is used in a piece.
```

ARGUMENTS:

```
- the sc-map reference
- the reference we're counting
```

RETURN VALUE:

 an integer reflecting the number of occurences of <ref>

SYNOPSIS:

```
(defmethod count-ref ((scm sc-map) ref)
```

28.5.376   sc-map/delete-nth-in-map

[sc-map] [Methods]

DATE:

 05 Feb 2011

DESCRIPTION:

 Replace the element at the specified location within the specified list of
 a given sc-map object with NIL.

ARGUMENTS:

 - A list that is the map-ref; i.e., the path of IDs into the list to be
   searched.
 - An integer that is the zero-based index of the element to be returned
   from the specified list.
 - An sc-map object.

RETURN VALUE:

 Always returns NIL

EXAMPLE:

```
(let ((mscm (make-sc-map 'scm-test
                         '((1
                            ((vn (1 2 3 4 5))
                             (va (2 3 4 5 1))
                             (vc (3 4 5 1 2))))
                           (2
                            ((vn (6 7 8))
                             (va (7 8 6))
                             (vc (8 6 7))))
```

```
                              (3
                               ((vn (9))
                                (va (9))
                                (vc (9)))))))))
  (delete-nth-in-map '(1 vn) 1 mscm)
  (get-data-from-palette '(1 vn) mscm))

=>
NAMED-OBJECT: id: VN, tag: NIL,
data: (1 NIL 3 4 5)
```

SYNOPSIS:

```
(defmethod delete-nth-in-map (map-ref nth (scm sc-map))
```

28.5.377   sc-map/double

[sc-map] [Methods]

DATE:

 13-Feb-2011

DESCRIPTION:

 Change the specified sequences of one or more specified players within an
 existing sc-map object to double the rhythms of the corresponding sequences
 of another specified player.

 This allows an existing map, for example, to have several players playing
 in rhythmic unison.

ARGUMENTS:

 - An sc-map object.
 - A section reference (i.e. section ID or list of section-subsection IDs).
 - An integer that is the 1-based number of the first sequence within the
   given section to be changed.
 - An integer that is the 1-based number of the last sequence within the
   given section to be changed.
 - The ID of the player whose part is to serve as the source for the
   doubling.
 - An ID or list of IDs of the player(s) whose parts are to be changed.

RETURN VALUE:

 Returns NIL.

EXAMPLE:

```
;;; Create an sc-map with parts for players 'fl and 'cl containing only NILs
;;; and print the corresponding data. Double the second and third sequence of
;;; the 'vn part of that section into the 'fl and 'cl parts and print the same
;;; data again to see the change.
(let ((scm (make-sc-map 'sc-m
                        '((1
                           ((fl (nil nil nil))
                            (cl (nil nil nil))
                            (vn (set1 set3 set2))
                            (va (set2 set3 set1))
                            (vc (set3 set1 set2))))
                          (2
                           ((vn (set1 set2 set1))
                            (va (set2 set1 set3))
                            (vc (set1 set3 set3))))
                          (3
                           ((vn (set1 set1 set3))
                            (va (set1 set3 set2))
                            (vc (set3 set2 set3)))))))))
  (print (get-data-data '(1 fl) scm))
  (print (get-data-data '(1 cl) scm))
  (double scm 1 2 3 'vn '(fl cl))
  (print (get-data-data '(1 fl) scm))
  (print (get-data-data '(1 cl) scm)))

=>
(NIL NIL NIL)
(NIL NIL NIL)
(NIL SET3 SET2)
(NIL SET3 SET2)
```

SYNOPSIS:

```
(defmethod double ((scm sc-map) section-ref start-seq end-seq master-player
                   doubling-players)
```

28.5.378   sc-map/get-all-data-from-palette

[sc-map] [Methods]

DESCRIPTION:

Given an sc-map object that has been bound to a palette object of any type,
return all of the palette data contained in the given sc-map object as it
has been allocated to the map, in the order in which it appears in the map.

The given sc-map object must be bound to a palette object for this method
to work. If no palette object has been bound to the given sc-map object,
the method returns NIL and prints a warning.

ARGUMENTS:

 - An sc-map object.

RETURN VALUE:

 - A list of objects, the type depending on the given palette.

EXAMPLE:

```
;; Create a set-palette object and an sc-map object, bind them using the
;; <palette> argument of the make-sc-map function, and print the results of
;; applying the get-all-data-from-palette method by printing the data of each
;; of the objects in the list it returns as note-name symbols.
(let* ((sp (make-set-palette 'set-pal '((set1 ((c2 b2 a3 g4 f5 e6)))
                                        (set2 ((d2 c3 b3 a4 g5 f6)))
                                        (set3 ((e2 d3 c4 b4 a5 g6))))))
      (scm (make-sc-map 'sc-m '((sec1
                                 ((vn (set1 set3 set2))
                                  (va (set2 set3 set1))
                                  (vc (set3 set1 set2))))
                                (sec2
                                 ((vn (set1 set2 set1))
                                  (va (set2 set1 set3))
                                  (vc (set1 set3 set3))))
                                (sec3
                                 ((vn (set1 set1 set3))
                                  (va (set1 set3 set2))
                                  (vc (set3 set2 set3)))))
                        :palette sp)))
  (loop for cs in (get-all-data-from-palette scm)
     collect (pitch-list-to-symbols (data cs))))

=>
((C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6) (D2 C3 B3 A4 G5 F6)
 (D2 C3 B3 A4 G5 F6) (E2 D3 C4 B4 A5 G6) (C2 B2 A3 G4 F5 E6)
 (E2 D3 C4 B4 A5 G6) (C2 B2 A3 G4 F5 E6) (D2 C3 B3 A4 G5 F6)
```

```
(C2 B2 A3 G4 F5 E6) (D2 C3 B3 A4 G5 F6) (C2 B2 A3 G4 F5 E6)
(D2 C3 B3 A4 G5 F6) (C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6)
(C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6) (E2 D3 C4 B4 A5 G6)
(C2 B2 A3 G4 F5 E6) (C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6)
(C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6) (D2 C3 B3 A4 G5 F6)
(E2 D3 C4 B4 A5 G6) (D2 C3 B3 A4 G5 F6) (E2 D3 C4 B4 A5 G6))

;; Applying the method to an sc-map object that is not bound to a palette
;; object returns NIL
(let ((scm (make-sc-map 'sc-m '((sec1
                                  ((vn (set1 set3 set2))
                                   (va (set2 set3 set1))
                                   (vc (set3 set1 set2))))
                                 (sec2
                                  ((vn (set1 set2 set1))
                                   (va (set2 set1 set3))
                                   (vc (set1 set3 set3))))
                                 (sec3
                                  ((vn (set1 set1 set3))
                                   (va (set1 set3 set2))
                                   (vc (set3 set2 set3))))))))
  (get-all-data-from-palette scm))

=>
NIL
WARNING:
   sc-map::get-all-data-from-palette:
   palette slot is nil so can't return data from it.
```

SYNOPSIS:

```
(defmethod get-all-data-from-palette ((scm sc-map))
```

28.5.379   sc-map/get-data-from-palette

[sc-map] [Methods]

DESCRIPTION:

Given an sc-map object that has been bound to a palette object of any type,
return the palette data contained at the location within the given sc-map
object as specified by the <IDs> argument.

Deeper levels of the map can be accessed by specifying a path of IDs into
the given sc-map object.

If no palette object has been bound to the given sc-map object, the method
returns the contents of the sc-map object at the specified location
instead.

ARGUMENTS:

- A symbol or list of symbols that is/are the ID or path of nested IDs
  within the given sc-map object for which the data is sought.
- The sc-map object in which the data is sought.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to print a warning if the specified ID is
  not found in the given sc-map object. T = print warning. Default = T.

RETURN VALUE:

The named object or list of named objects associated with the specified ID
or path of IDs.

If the specified ID is not found within the given sc-map object, the method
returns NIL. If the optional <warn> argument is set to T, a warning is also
printed in this case.

EXAMPLE:

```
;;; Create a palette object and an sc-map object and bind them using the
;;; <palette> keyword argument of the make-sc-map function. Then apply the
;;; get-data-from-palette object to a nested ID in the sc-map object. Loop
;;; through the data of the named objects in the list returned and return them
;;; as note-name symbols.
(let* ((sp (make-set-palette 'set-pal '((set1 ((c2 b2 a3 g4 f5 e6)))
                                        (set2 ((d2 c3 b3 a4 g5 f6)))
                                        (set3 ((e2 d3 c4 b4 a5 g6))))))
       (scm (make-sc-map 'sc-m '((sec1
                                  ((vn (set1 set3 set2))
                                   (va (set2 set3 set1))
                                   (vc (set3 set1 set2))))
                                 (sec2
                                  ((vn (set1 set2 set1))
                                   (va (set2 set1 set3))
                                   (vc (set1 set3 set3))))
                                 (sec3
                                  ((vn (set1 set1 set3))
```

```
                                        (va (set1 set3 set2))
                                        (vc (set3 set2 set3)))))
                           :palette sp)))
  (loop for cs in (get-data-from-palette '(sec1 vn) scm)
     collect (pitch-list-to-symbols (data cs))))

=> ((C2 B2 A3 G4 F5 E6) (E2 D3 C4 B4 A5 G6) (D2 C3 B3 A4 G5 F6))

;; If applied to an sc-map object that is not bound to a palette, the contents
;; of the sc-map object at the specified location are returned and a warning is
;; printed by default
(let ((scm (make-sc-map 'sc-m '((sec1
                                  ((vn (set1 set3 set2))
                                   (va (set2 set3 set1))
                                   (vc (set3 set1 set2))))
                                 (sec2
                                  ((vn (set1 set2 set1))
                                   (va (set2 set1 set3))
                                   (vc (set1 set3 set3))))
                                 (sec3
                                  ((vn (set1 set1 set3))
                                   (va (set1 set3 set2))
                                   (vc (set3 set2 set3)))))))))
   (get-data-from-palette '(sec1 vn) scm))

=>
NAMED-OBJECT: id: VN, tag: NIL,
data: (SET1 SET3 SET2)
**************
, NO-PALETTE
```

SYNOPSIS:

```
(defmethod get-data-from-palette (ids (scm sc-map) &optional (warn t))
```

28.5.380   sc-map/get-nth-from-map

[sc-map] [Methods]

DESCRIPTION:

 Get the element located at the nth position within a given sc-map
 object. Both the map-ref (the path of IDs into the list to be searched) and
 the nth must be specified.

ARGUMENTS:

 - A list that is the map-ref; i.e., the path of IDs into the list to be
   searched.
 - An integer that is the zero-based index of the element to be returned
   from the specified list.
 - An sc-map object.

RETURN VALUE:

 Returns the element located at the given index.

 Returns NIL if the index does not exist.

EXAMPLE:

```
;; Specify the path of IDs into the desired list ("map-ref") as a list, then
;; the position to be read from within the list located there.
(let ((mscm (make-sc-map 'scm-test
                         '((1
                            ((vn (1 2 3 4 5))
                             (va (2 3 4 5 1))
                             (vc (3 4 5 1 2))))
                           (2
                            ((vn (6 7 8))
                             (va (7 8 6))
                             (vc (8 6 7))))
                           (3
                            ((vn (9))
                             (va (9))
                             (vc (9)))))))
  (get-nth-from-map '(1 vn) 1 mscm))

=> 2

;; Returns NIL if the specified index does not exist
(let ((mscm (make-sc-map 'scm-test
                         '((1
                            ((vn (1 2 3 4 5))
                             (va (2 3 4 5 1))
                             (vc (3 4 5 1 2))))
                           (2
                            ((vn (6 7 8))
                             (va (7 8 6))
                             (vc (8 6 7))))
                           (3
                            ((vn (9))
                             (va (9))
```

```
                                       (vc (9))))))))))
  (get-nth-from-map '(3 vn) 1 mscm))
```

=> NIL

SYNOPSIS:

```
(defmethod get-nth-from-map (map-ref nth (scm sc-map))
```

28.5.381   sc-map/get-nth-from-palette

[sc-map] [Methods]

DESCRIPTION:

 Given an sc-map object that is bound to a palette object of any type,
 return the data of the palette object located at the nth position of the
 list found at the specified ID or path of nested IDs.

 If the given sc-map object is not bound to a palette object, NIL is
 returned instead.

ARGUMENTS:

 - An ID or list of IDs that are the path to the list within the given
   sc-map object from which the specified nth position is to be returned.
 - A zero-based integer that is the position within the list found at the
   path specified from which the given element is to be returned.
 - An sc-map object.

RETURN VALUE:

 - An element/object of the type contained within the given palette object
   of the given sc-map object.

EXAMPLE:

```
;;; Create a set-palette object and an sc-map object, bind them using the
;;; <palette> object of the make-sc-map function, and apply the
;;; get-nth-from-palette method
(let* ((sp (make-set-palette 'set-pal '((set1 ((c2 b2 a3 g4 f5 e6)))
                                        (set2 ((d2 c3 b3 a4 g5 f6)))
                                        (set3 ((e2 d3 c4 b4 a5 g6)))))))
      (scm (make-sc-map 'sc-m '((sec1
                                  ((vn (set1 set3 set2))
```

```
                                         (va (set2 set3 set1))
                                         (vc (set3 set1 set2))))
                                    (sec2
                                     ((vn (set1 set2 set1))
                                      (va (set2 set1 set3))
                                      (vc (set1 set3 set3))))
                                    (sec3
                                     ((vn (set1 set1 set3))
                                      (va (set1 set3 set2))
                                      (vc (set3 set2 set3)))))
                            :palette sp)))
  (get-nth-from-palette '(sec1 vn) 0 scm))


=>
COMPLETE-SET: complete: NIL
              num-missing-non-chromatic: 12
              num-missing-chromatic: 6
              missing-non-chromatic: (BQS BQF AQS AQF GQS GQF FQS EQS EQF DQS
                                         DQF CQS)
              missing-chromatic: (BF AF FS EF D CS)
TL-SET: transposition: 0
        limit-upper: NIL
        limit-lower: NIL
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 0
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 0, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: USED-NOTES, tag: NIL,
data: NIL
**************


**** N.B. All pitches printed as symbols only, internally they are all
pitch-objects.


    subsets:
    related-sets:
SCLIST: sclist-length: 6, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SET1, tag: NIL,
```

```
data: (C2 B2 A3 G4 F5 E6)
**************

;;; Applying the method to an sc-map object that is not bound to a palette
;;; object returns NIL
(let ((scm (make-sc-map 'sc-m '((sec1
                                  ((vn (set1 set3 set2))
                                   (va (set2 set3 set1))
                                   (vc (set3 set1 set2))))
                                 (sec2
                                  ((vn (set1 set2 set1))
                                   (va (set2 set1 set3))
                                   (vc (set1 set3 set3))))
                                 (sec3
                                  ((vn (set1 set1 set3))
                                   (va (set1 set3 set2))
                                   (vc (set3 set2 set3)))))))))
  (get-nth-from-palette '(sec1 vn) 0 scm))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-from-palette (sc-map-ref nth (scm sc-map))
```

28.5.382   sc-map/make-sc-map

[sc-map] [Functions]

DESCRIPTION:

 Create an sc-map object, which will be used for mapping rhythmic sequences,
 chords etc. to specific parts of a piece.

ARGUMENTS:

 - The ID of the resulting sc-map object.
 - A list of data, most likely recursive.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :warn-not-found. T or NIL to indicate whether a warning is printed when
   an index which doesn't exist is used for look-up. T = warn. Default = T.
 - :recurse-simple-data. T or NIL to indicate whether to recursively

instantiate a recursive-assoc-list in place of data that appears to be a
simple assoc-list (i.e. a 2-element list). If NIL, the data of 2-element
lists whose second element is a number or a symbol will be ignored,
therefore remaining as a list. For example, this data would normally
result in a recursive call: (y ((2 23) (7 28) (18 2))).
T = recurse. Default = T.
- :replacements. A list of lists in the format '(((1 2 vla) 3 20b) ((2 3
vln) 4 16a)) that indicate changes to individual elements of lists within
the given sc-map object. (Often sc-map data is generated algorithmically,
but individual elements of the lists need to be changed.) Each such list
indicates a change, the first element of the list being the reference
into the sc-map (the viola voice of section 1 subsection 2 in the first
element here, for example), the second element being the nth of the data
list to change for this key, and the third being the new data.
- :palette. A palette object or NIL. If a palette object is specified or
defined here, it will be automatically bound to the given sc-map
object. Default = NIL

RETURN VALUE:

 An sc-map object.

EXAMPLE:

```
;; Create an sc-map object with contents that could be used as a rthm-seq-map
(make-sc-map 'scm-test
             '((1
                ((vn (1 2 3 4 5))
                 (va (2 3 4 5 1))
                 (vc (3 4 5 1 2))))
               (2
                ((vn (6 7 8))
                 (va (7 8 6))
                 (vc (8 6 7))))
               (3
                ((vn (9))
                 (va (9))
                 (vc (9))))))

=>
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 9
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
```

```
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SCM-TEST, tag: NIL,
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 3
                      linked: NIL
                      full-ref: (1)
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "sub-ral-of-SCM-TEST", tag: NIL,
data: (
NAMED-OBJECT: id: VN, tag: NIL,
data: (1 2 3 4 5)
[...]
```

```
;;; Create an sc-map object and automatically bind it to a set-palette object
;;; using the <palette> keyword argument. Then read the PALETTE slot of the
;;; sc-map created to see its contents.
(let ((scm
       (make-sc-map
        'scm-test
        '((1
           ((vn (1 2 3 4 5))
            (va (2 3 4 5 1))
            (vc (3 4 5 1 2))))
          (2
           ((vn (6 7 8))
            (va (7 8 6))
            (vc (8 6 7))))
          (3
           ((vn (9))
            (va (9))
            (vc (9)))))
        :palette (make-set-palette 'set-pal
                                   '((set1 ((c2 b2 a3 g4 f5 e6)))
                                     (set2 ((d2 c3 b3 a4 g5 f6)))
                                     (set3 ((e2 d3 c4 b4 a5 g6)))))))))
  (palette scm))

=>
```

```
SET-PALETTE:
PALETTE:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 3
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SET-PAL, tag: NIL,
data: (
COMPLETE-SET: complete: NIL
[...]
data: (C2 B2 A3 G4 F5 E6)
[...]
COMPLETE-SET: complete: NIL
[...]
data: (D2 C3 B3 A4 G5 F6)
[...]
COMPLETE-SET: complete: NIL
[...]
data: (E2 D3 C4 B4 A5 G6)
)

;;; An example using replacements
(make-sc-map 'sc-m
             '((1
                ((vn (set1 set3 set2))
                 (va (set2 set3 set1))
                 (vc (set3 set1 set2))))
               (2
                ((vn (set1 set2 set1))
                 (va (set2 set1 set3))
                 (vc (set1 set3 set3))))
               (3
                ((vn (set1 set1 set3))
                 (va (set1 set3 set2))
                 (vc (set3 set2 set3)))))
             :replacements '(((1 va) 2 set2)))

=>

SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
```

```
                        num-data: 9
                        linked: NIL
                        full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SC-M, tag: NIL,
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                        num-data: 3
                        linked: NIL
                        full-ref: (1)
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "sub-ral-of-SC-M", tag: NIL,
data: (
NAMED-OBJECT: id: VN, tag: NIL,
data: (SET1 SET3 SET2)
**************


NAMED-OBJECT: id: VA, tag: NIL,
data: (SET2 SET2 SET1)
**************


NAMED-OBJECT: id: VC, tag: NIL,
data: (SET3 SET1 SET2)
**************
)

[...]
```

SYNOPSIS:

```
(defun make-sc-map (id scm &key (palette nil) (warn-not-found t)
                    (recurse-simple-data t) (replacements nil))
```

28.5.383   sc-map/num-sequences

[sc-map] [Methods]

DATE:

 January 27th 2018

DESCRIPTION:

 Return the number of sequences in the map as a whole (i.e. counting all
 sections and subsections). This method also works on subsections. Note that
 the first time this method is used it sets the :num-sequences slot which
 will then be used in subsequent queries until the data is changed. However,
 if you've not specifically called this method on a subsection, its slot will
 not yet be set.

ARGUMENTS:

 - an sc-map object

RETURN VALUE:

 An integer

SYNOPSIS:

(defmethod num-sequences ((scm sc-map))

28.5.384   sc-map/remove-ref

[sc-map] [Methods]

DATE:

 August 24th 2018

DESCRIPTION:

 Remove or replace a reference from a map. If there's no replacement then the
 data list will be shorter (if the reference is present) thereafter (which
 could cause problems .

ARGUMENTS:

- an sc-map object
- the section reference (symbol, list, etc.)
- the reference in the data list that should be removed

OPTIONAL ARGUMENTS:

- the replacement reference, if any. If, for example in a rthm-seq-map, you
  want to replace a rthm-seq reference with nil, so that the player sits out
  a sequence, then pass '(nil) as the replacement.

RETURN VALUE:

 The sc-map object.

SYNOPSIS:

(defmethod remove-ref ((scm sc-map) section ref &optional replacement)

28.5.385   sc-map/rthm-seq-map

[sc-map] [Classes]

NAME:

 rthm-seq-map

 File: rthm-seq-map.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                   circular-sclist -> assoc-list -> recursive-assoc-list ->
                   sc-map -> rthm-seq-map

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the rthm-seq-map class which maps
                   references to rthm-seq objects for the players in the
                   piece. Extensions to the sc-map superclass are the
                   collection of all the players in the piece and a
                   check to make sure that each list each instrument has the
                   same number of rthm-seq references for each section.

                   Instances of this class must declare sections and
                   players so if the piece is in one section, give it

```
                the label 1 or whatever, e.g.

                '((1
                   ((vln (2 20 1 9 10 22 16 25 6 14 21 17 4 9 13 2))
                    (vla (2 23 3 7 13 22 19 3 8 12 23 14 2 10 15 4))
                    (vc (2 21 3 12 11 22 16 1 8 17 23 20 24 9 12 2)))))
```

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: July 28th 2001

 $$ Last modified: 19:19:54 Tue Jul 14 2020 CEST

 SVN ID: Id

28.5.386   rthm-seq-map/add-player

[rthm-seq-map] [Methods]

DATE:

 January 18th 2018, Heidhausen

DESCRIPTION:

 Add a player to a rthm-seq-map. The data (3rd) argument defines the rthm-seq
 references (IDs) to be used for the new player. This is a very flexible
 argument:

 If not given, then NIL will be added to the map, resulting in the new player
 not playing at all.

 If a single reference ID is given, then that will be used throughout each
 section of the map.

 If a list of references is given, then items will be popped out of the list
 sequentially and used for the new player. In this case the length of the
 list does not matter as if it's too long the last elements will simply be
 unused, and if it's too short then NIL will be returned when the list is
 exhausted, resulting in the new player not playing from that point
 onwards. The exception here is if the cycle (4th) argument is T, whereupon
 the list of references will be used circularly, restarting the list when the
 end is reached.

Otherwise a function can be passed: this function should take four
arguments: the rthm-seq-map object (which it can then analyse/query to make
decisions); the full reference to the current (sub)section for which
references should be provided; the number of references which are needed
for this (sub)section; and the player being added. It must return a list of
references for each given section.

ARGUMENTS:

- the rthm-seq-map object
- the new player to add (ID only: the same which will be referenced in the
  ensemble slot of a slippery-chicken object)

OPTIONAL ARGUMENTS:

- the rthm-seq-palette references to use for the new player. See above.
  Default NIL.
- T or NIL to cycle the references. See above. Default NIL.

RETURN VALUE:

the same rthm-seq-map object but with the new player now added.

EXAMPLE:

```
;; will use rthm-seq 1 throughout the piece
(add-player m 'cl 1)
;; will use 1 2 3 4 5 then nil thereafter
(add-player m 'ob '(1 2 3 4 5))
;; forcing a reference into a recursive rthm-seq-palette to be used just once
(add-player m 'db '((bass dense 1)))
;; forcing it to be used repeatedly
(add-player m 'db '((bass dense 1)) t)
;; a simple example with a function: rs1 for section 1, rs3 for section three
;;; otherwise rs1a
(add-player m 'bsn #'(lambda (rsm section-ref num-refs player)
                          (ml (case section-ref
                                (sec1 'rs1)
                                (sec3 'rs3)
                                (t 'rs1a))
                              num-refs)))
;; db will be added but it won't play anything (unless references are added
;;; later but before make-slippery-chicken is called)
(add-player m 'db)
```

SYNOPSIS:

```
(defmethod add-player ((rsm rthm-seq-map) player &optional data cycle
                                          ignore1 ignore2)
```

28.5.387   rthm-seq-map/add-repeats

[rthm-seq-map] [Methods]

DATE:

 30-Dec-2010

DESCRIPTION:

 Generate repeating sequences at given cycle points using recurring-event
 data. This extends the number of items in the map.

ARGUMENTS:

 - A rthm-seq-map object.
 - A list of two-item lists of integers that determine the cycle
   pattern. This list will have the format of the recurring-event class's
   DATA slot (see recurring-event.lsp).
 - A list of two-item lists of integers that determine the number of repeats
   made (or references into the :repeats list). This list is also processed
   cyclically (i.e. the recurring-event class's RETURN-DATA-CYCLE slot).

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :section. The section map reference. Default = 1.
 - :repeats-indices. A list of the number of repeat bars returned by the
    cycle data (i.e. recurring-event class's RETURN-DATA slot). Generally
    this will remain NIL and the number of repeats will be expressed
    directly in the third argument, but it could be useful to use references
    into this list there instead, since the recurring-event class already
    makes this possible. Default = NIL.
 - :start. An integer that is the number of the bar/rthm-seq where the
   process is to begin. Default = 1.
 - :end. An integer that is the number of the bar/rthm-seq where the process
   is to end. NIL = process all bars/rthm-seqs. Default = NIL.
 - :print. T or NIL to indicate whether to print the rthm-seq ID and the
   number repetitions to the listener. T = print. Default = NIL.
 - :repeat-rest-seqs. T or NIL to indicate whether sequences consisting of
   just rests should be repeated also. This will need the :palette in order
   to work. Default = T.

  - :palette. The rthm-seq-palette to check that sequences are not
    rest sequences, if about to repeat them and repeat-rest-seqs is NIL.

RETURN VALUE:

 An integer that is the number of sequences added.

EXAMPLE:

```
;;; Straightforward usage, additionally printing the DATA slot before and after
;;; applying the method
(let ((mrsm
       (make-rthm-seq-map
        'rsm-test
        '((1 ((vn (1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)))))
        :palette (make-rsp
                   'rs-pal
                   '((1 ((((2 4) q e s s))))
                     (2 ((((2 4) e s s q))))
                     (3 ((((2 4) s s q e)))))))))
  (print (get-data-data '(1 vn) mrsm))
  ;; so there'll be a repeat after two events three times in a row, then after
  ;; three events twice in a row. The number of repeats will be 5 eight times
  ;; in a row, then 8 twice in a row.
  (add-repeats mrsm '((2 3) (3 2)) '((5 3) (8 2)))
  (print (get-data-data '(1 vn) mrsm)))

=>
(1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)
(1 2 3 3 3 3 3 2 1 1 1 1 1 3 1 1 1 1 1 3 2 3 3 3 3 3 3 3 3 1 2 1 1 1 1 1 1
 1 1 3 1 1 1 1 1 3 2 2 2 2 2 2 1)

;;; Using the :start, :end, and :print arguments
(let ((mrsm
       (make-rthm-seq-map
        'rsm-test
        '((1 ((vn (1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)))))
        :palette (make-rsp
                   'rs-pal
                   '((1 ((((2 4) q e s s))))
                     (2 ((((2 4) e s s q))))
                     (3 ((((2 4) s s q e)))))))))
  (print (get-data-data '(1 vn) mrsm))
  (add-repeats mrsm '((1 6) (2 6)) '((11 6) (23 3))
               :start 3
               :end 11
```

```
                :print t)
  (print (get-data-data '(1 vn) mrsm)))

=>
(1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)
2 x 11
1 x 11
3 x 11
1 x 11
3 x 11
2 x 11
1 x 23
(1 2 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 1
 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 3 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 3 1 3 2 1)
```

SYNOPSIS:

```
(defmethod add-repeats ((rsm rthm-seq-map) repeat-every repeats &key
                        (repeat-rest-seqs t) palette
                        (section 1) repeats-indices (start 1) end print)
```

28.5.388   rthm-seq-map/add-repeats-simple

[rthm-seq-map] [Methods]

DESCRIPTION:

 Add repeats of a specified rthm-seq within the given rthm-seq-map object a
 specified number of times.

ARGUMENTS:

 - A rthm-seq-map object.
 - An integer that is the number of the rthm-seq (position within the
   rthm-seq-map) to be repeated.
 - An integer that is the number of times that rthm-seq is to be repeated.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :section. An integer that is the ID of the section in which the repeat
   operation is to be performed.
 - :print. T or NIL to indicate whether to print the rthm-seq ID and the
   number repetitions to the listener. T = print. Default = NIL.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;;; Print the DATA of the given rthm-seq-map, apply the method, and print again
;;; to see the difference.
(let ((mrsm
       (make-rthm-seq-map
        'rsm-test
        '((1 ((vn (1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)))))
        :palette (make-rsp
                  'rs-pal
                  '((1 ((((2 4) q e s s))))
                    (2 ((((2 4) e s s q))))
                    (3 ((((2 4) s s q e)))))))))
  (print (get-data-data '(1 vn) mrsm))
  (add-repeats-simple mrsm 3 13)
  (print (get-data-data '(1 vn) mrsm)))

=>
(1 2 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)
(1 2 3 3 3 3 3 3 3 3 3 3 3 3 3 2 1 3 1 3 2 3 1 2 1 3 1 3 2 1)
```

SYNOPSIS:

```
(defmethod add-repeats-simple ((rsm rthm-seq-map) start-seq repeats &key
                                   (section 1) print)
```

28.5.389   rthm-seq-map/check-num-sequences

[rthm-seq-map] [Functions]

DESCRIPTION:

 Check to ensure that each player in each section of the given rthm-seq-map
 object has the same number of references as every other instrument. If not,
 drop into the debugger with an error.

 NB: This function is called automatically every time make-rthm-seq-map is
     called so it shouldn't generally be necessary for the user to call this
     method. However, if the rthm-seq-map is changed somehow, it might be a
     good idea to recheck.

ARGUMENTS:

- A rthm-seq-map object.

RETURN VALUE:

Returns T if all players have the same number of references in each
section, otherwise drops into the debugger with an error.

EXAMPLE:

```
;;; Passes the test:
(let ((rsmt (make-rthm-seq-map
             'rsm-test
             '((sec1 ((vn (rs1a rs3a rs2a))
                      (va (rs1b rs3b rs2b))
                      (vc (rs1a rs3b rs2a))))
               (sec2 ((vn (rs1a rs2a rs1a))
                      (va (rs1a rs2a rs1b))
                      (vc (rs1a rs2b rs1a))))
               (sec3 ((vn (rs1a rs1a rs3a))
                      (va (rs1a rs1a rs3b))
                      (vc (rs1a rs1b rs3a))))
               (sec4 ((vn (rs1a rs1a rs1a))
                      (va (rs1a rs1a rs1b))
                      (vc (rs1a rs1b rs1a)))))))))
  (check-num-sequences rsmt))

=> T

;;; Doesn't pass the test; drops into debugger with an error.
(let ((rsmt (make-rthm-seq-map
             'rsm-test
             '((sec1 ((vn (rs1a rs3a rs2a))
                      (va (rs1b rs3b))
                      (vc (rs1a rs3b rs2a))))
               (sec2 ((vn (rs1a))
                      (va (rs1a rs2a rs1b))
                      (vc (rs1a rs2b rs1a))))
               (sec3 ((vn (rs1a rs3a))
                      (va (rs1a))
                      (vc (rs1a rs1b rs3a))))
               (sec4 ((vn (rs1a))
                      (va (rs1a rs1a rs1b))
                      (vc (rs1a rs1a)))))))))
  (check-num-sequences rsmt))

=>
```

```
rthm-seq-map::check-num-sequences: In rthm-seq-map RSM-TEST-5, instrument VA:
Each instrument must have the same number of sequences for any given section:
(RS1B RS3B)
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defun check-num-sequences (rsm)
```

28.5.390   rthm-seq-map/get-map-refs

[rthm-seq-map] [Methods]

DATE:

```
 29-Dec-2010
```

DESCRIPTION:

```
 Return the list of rthm-seq-palette references for the given player and
 section.
```

ARGUMENTS:

```
 - A rthm-seq-map object.
 - The ID of the section in which the references are sought.
 - The ID of the player for whom the references are sought.
```

RETURN VALUE:

```
 A list of references (each of which might also  be a list).
```

EXAMPLE:

```
(let ((rsmt (make-rthm-seq-map
              'rsm-test-5
              '((sec1 ((vn (rs1 rs3 rs2))
                       (va (rs2 rs3 rs1))
                       (vc (rs3 rs1 rs2))))
                (sec2 ((vn (rs1 rs2 rs1))
                       (va (rs2 rs1 rs3))
                       (vc (rs1 rs3 rs3))))
                (sec3 ((vn (rs1 rs1 rs3))
                       (va (rs1 rs3 rs2))
```

```
                          (vc (rs3 rs2 rs3)))))
             :palette (make-rsp
                       'rs-pal
                      '((rs1 ((((2 4) q e s s))))
                        (rs2 ((((2 4) e s s q))))
                        (rs3 ((((2 4) s s q e)))))))))))
  (get-map-refs rsmt 'sec3 'vc))
```

=> (RS3 RS2 RS3)

SYNOPSIS:

```
(defmethod get-map-refs ((rsm rthm-seq-map) section player)
```

28.5.391   rthm-seq-map/get-time-sig-ral

[rthm-seq-map] [Methods]

DESCRIPTION:

Collate the IDs of all rthm-seq objects in a given rthm-seq-map into groups
based on identical time signatures and return this as a
recursive-assoc-list object that has the same structure of the map.

Instead of having the list of rthm-seq references for each
section/instrument, the method returns an assoc-list whose keys are the
time-sig tags, with the corresponding data being circular-sclists of
rthm-seq IDs.

The result is a recursive-assoc-list for each instrument/section that can
be queried to find the rthm-seq refs of all rthm-seqs that share the same
bar/time-sig structure; e.g., all those that have a 2/4 bar followed by a
3/4 bar etc.

ARGUMENTS:

 - A rthm-seq-map object.
 - A rthm-seq-palette object.

RETURN VALUE:

 A recursive-assoc-list object.

EXAMPLE:

```
(let* ((mini
         (make-slippery-chicken
          '+mini+
          :ensemble '(((sax (alto-sax :midi-channel 1))))
          :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
          :set-map '((1 (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)))
          :rthm-seq-palette '((1 ((((4 4) h q e s s) ((2 4) h))))
                              (2 ((((4 4) h q e s s))))
                              (3 ((((4 4) h q e s s))))
                              (4 ((((4 4) h q q) ((2 4) q q))))
                              (5 ((((4 4) h q e s s))))
                              (6 ((((4 4) h q q) ((2 4) q q)))))
          :rthm-seq-map '((1 ((sax (1 2 3 5 2 4 6 2 3 1 3 2 3 2 1 3 2)))))))
       (tsral (get-time-sig-ral (rthm-seq-map mini) (rthm-seq-palette mini))))
  (get-data-data 1 tsral))

=>
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 1
                      linked: T
                      full-ref: (1)
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 1, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "sub-ral-of-+MINI+-RTHM-SEQ-MAP", tag: NIL,
data: (
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: (1 SAX), next: NIL
NAMED-OBJECT: id: SAX, tag: NIL,
data: (
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0404-0204", tag: NIL,
data: (4 6 1)
**************


CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0404", tag: NIL,
data: (5 3 2)
```

```
**************
)
**************
)
**************
```

SYNOPSIS:

```
(defmethod get-time-sig-ral ((rsm rthm-seq-map) (rsp rthm-seq-palette))
```

28.5.392   rthm-seq-map/make-rthm-seq-map

[rthm-seq-map] [Functions]

DESCRIPTION:

 Make a rthm-seq-map object.

ARGUMENTS:

```
 - The ID of the rthm-seq-map object to be made.
 - A list of nested lists, generally taking the form
   '((section1 ((player1 (rthm-seq ids))
               (player2 (rthm-seq ids))
               (etc... (etc...))))
     (section2 ((player1 (rthm-seq ids))
               (player2 (rthm-seq ids))
               (etc...)))
     (etc...))
```

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :palette. A palette object or NIL. If a palette object is specified or
   defined here, it will be automatically bound to the given rthm-seq-map
   object. Default = NIL.
 - :warn-not-found. T or NIL to indicate whether a warning is printed when
   an index which doesn't exist is used for look-up.
   T = warn. Default = NIL.
 - :replacements. A list of lists in the format
   '(((1 2 vla) 3 20b) ((2 3 vln) 4 16a)) that indicate changes to
   individual elements of lists within the given rthm-seq-map object. (Often
   rthm-seq-map data is generated algorithmically but individual elements of
   the lists need to be changed.)  Each such list indicates a change, the
   first element of the list being the reference into the rthm-seq-map (the
```

   vla player of section 1, subsection 2 in the first example here), the
   second element is the nth of the data list for this key to change, and
   the third is the new data. Default = NIL.
 - :recurse-simple-data. T or NIL to indicate whether to recursively
   instantiate a recursive-assoc-list in place of data that appears to be a
   simple assoc-list (i.e. a 2-element list). If NIL, the data of 2-element
   lists whose second element is a number or a symbol will be ignored,
   therefore remaining as a list. For example, this data would normally
   result in a recursive call: (y ((2 23) (7 28) (18 2))).
   T = recurse. Default = T.

RETURN VALUE:

 A rthm-seq-map object.

EXAMPLE:

```
;;; Straightforward usage
(make-rthm-seq-map 'rsm-test
                    '((1 ((vn (1 2 3 4))
                          (va (2 3 4 1))
                          (vc (3 4 1 2))))
                      (2 ((vn (4 5 6))
                          (va (5 6 4))
                          (vc (6 4 5))))
                      (3 ((vn (7 8 9 1 2))
                          (va (8 9 1 2 7))
                          (vc (9 1 2 7 8)))))))

=>

RTHM-SEQ-MAP: num-players: 3
              players: (VA VC VN)
SC-MAP: palette id: NIL
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 9
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: RSM-TEST, tag: NIL,
data: (
[...]
```

```
;;; An example using the :replacements argument and binding directly to a
;;; specified rthm-seq-palette object.
(make-rthm-seq-map 'rsm-test
                   '((1 ((vn (1 2 3 4))
                         (va (2 3 4 1))
                         (vc (3 4 1 2))))
                     (2 ((vn (4 5 6))
                         (va (5 6 4))
                         (vc (6 4 5))))
                     (3 ((vn (7 8 9 1 2))
                         (va (8 9 1 2 7))
                         (vc (9 1 2 7 8)))))
                   :palette (make-rsp
                             'rs-pal
                             '((rs1 ((((2 4) q e s s))))
                               (rs2 ((((2 4) e s s q))))
                               (rs3 ((((2 4) s s q e))))))
                   :replacements '(((1 vn) 2 7)
                                   ((2 va) 1 1)
                                   ((3 vc) 1 0)))

=>
RTHM-SEQ-MAP: num-players: 3
              players: (VA VC VN)
SC-MAP: palette id: RS-PAL
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 9
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: RSM-TEST, tag: NIL,
data: (
[...]
```

SYNOPSIS:

```
(defun make-rthm-seq-map (id rsm &key (palette nil) (warn-not-found nil)
                                      (replacements nil)
                                      (recurse-simple-data t))
```

28.5.393   rthm-seq-map/rsm-count-notes

[rthm-seq-map] [Functions]

DESCRIPTION:

 Returns the number of notes in the given rthm-seq-map object for the
 specified player and rthm-seq-palette.

ARGUMENTS:

 - A rthm-seq-map object.
 - The ID of the player whose notes are to be counted.
 - The rthm-seq-palette object whose rthm-seq object IDs are referred to by
   the given rthm-seq-map object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to count just the number of notes that need
   new events (i.e., not counting tied notes; also not counting chords,
   since chords need only one event) or the total number of notes in that
   player's part in the score. T = count just attacked notes. Default = T.

RETURN VALUE:

 Returns an integer that is the number of notes counted.

EXAMPLE:

```
(let ((rsmt (make-rthm-seq-map
             'rsm-test
             '((sec1 ((vn (rs1 rs3 rs2))
                      (va (rs2 rs3 rs1))
                      (vc (rs3 rs1 rs2))))
               (sec2 ((vn (rs1 rs2 rs1))
                      (va (rs2 rs1 rs3))
                      (vc (rs1 rs3 rs3))))
               (sec3 ((vn (rs1 rs1 rs3))
                      (va (rs1 rs3 rs2))
                      (vc (rs3 rs2 rs3)))))))
      (rspt (make-rsp
             'rs-pal
             '((rs1 ((((2 4) q (e) s s))))
               (rs2 ((((2 4) e +s (s) q))))
               (rs3 ((((2 4) (s) s +q e)))))))))
```

```
  (print (rsm-count-notes rsmt 'vn rspt))
  (print (rsm-count-notes rsmt 'va rspt nil)))

=>
23
27
```

SYNOPSIS:

```
(defun rsm-count-notes (rthm-seq-map player palette &optional (just-attacks t))
```

28.5.394   rthm-seq-map/rthm-chain

[rthm-seq-map] [Classes]

NAME:

```
 rthm-chain

 File:          rthm-chain.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> recursive-assoc-list ->
                  sc-map -> rthm-seq-map -> rthm-chain

 Version:       1.0.11

 Project:       slippery chicken (algorithmic composition)

 Purpose:       Algorithmic generation of rthm-seqs that include
                slower-moving counterpoint and a means to control
                activity development through curves.  Here we generate a
                rthm-seq-map and its associated palette algorithmically.

                Say we have 9 irregular 1 beat duration patterns; these
                would enter in the sequence defined by (procession x 9)
                where x would be the number of patterns to generate.

                Rests are inserted at regular but changing intervals e.g

                3x every 2 beats (6)
                2x every 3 beats (6)
                3x every 5 beats (15)
                2x every 8 beats (16)
```

e, q, and q. rests are used by default, in a sequence
determined by a recurring-event instance.

In order to make music that 'progresses' we have curves
with y values from 1-10 indicating how much activity
there should be: 1 would mean only 1 in 10 beats would
have notes in/on them, 10 would indicate that all do. We
use the patterns given in
activity-levels::initialize-instance, where 1 means
'play', 0 means 'rest'. There are three examples of each
level so that if we stick on one level of activity for
some time we won't always get the same pattern: these
will instead be cycled through.

A slower moving (bass) line is also added that is made up
of 2 or 3 beat groups---if the activity curve indicates a
rest, then the whole 2-3 beat group is omitted.

There are also 'sticking points' where a rhythm will be
repeated a certain number of times (either s, e, e., or q
by default). Sticking happens after rests. This can be
controlled with an activity envelope too, also indicating
one of the 10 patterns above (but also including 0). A 0
or 1 unit here would refer to a certain number of repeats
(1) or none (0). How many repeats could be determined by
something like: (procession 34 '(2 3 5 8 13) :peak 1
:expt 3) There's always a slower group to accompany the
sticking points: simply the next in the sequence,
repeated for as long as we stick

The harmonic-rthm curve specifies how many slower-rthms
will be combined into a rthm-seq (each rthm-seq has a
single harmony). The default is 2 bars (slower-rthms)
per rthm-seq.

Author: Michael Edwards: m@michael-edwards.org

Creation date: 4th February 2010

$$ Last modified: 16:14:15 Thu Sep 17 2020 CEST

SVN ID: Id

28.5.395   rthm-chain/add-voice

[rthm-chain] [Methods]

DESCRIPTION:

Add a new voice to an existing rthm-chain object based on the rhythmic
material and slot values already contained in that object.

The main rthm-chain algorithm generates only two voices. Rather than
generate further voices in the same fashion by which the first two were
created, this method uses the already created rthm-seqs in the given
rthm-chain object to create a new voice.

The challenge here is that each rthm-seq potentially has its own time
signature structure: There could be a 2/4 bar followed by 5/4 then 3/16,
for example, or any other combination of any meter. So the method first
analyses the time-signature structure of the existing rthm-seqs and saves
those with the same bar/meter structure together in the order in which they
occur. When creating the extra voice then, the method actually starts ahead
of the main voice by choosing <offset> number of similar rthm-seqs in
advance. NB Your data might well produce only one rthm-seq with a
particular metric structure, which might mean that calling this method
produces rhythmic doubling instead of an independent part. In that case
you might try adding more rhythmic fragments and regenerating your piece.

ARGUMENTS:

- A rthm-chain object.
- The reference (key path) of the player within the given rthm-chain object
  whose rthm-seq-map is to serve as the 'parent voice', e.g. '(1 cl).
- A symbol that will be the ID of the new player.

OPTIONAL ARGUMENTS:

- An integer that indicates an offset into the group of similar rthm-seq
  objects from which the new voice is to begin. (The generated voice will
  thus be ahead of the main voice). Default = 1.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((rch
```

```
    (make-rthm-chain
     'test-rch 150
     '((((e) e) ; 4 in total
        (- s (s) (s) s -)
        ({ 3 (te) - te te - })
        ((e.) s))
       (({ 3 (te) te (te) }) ; what we transition to
        ({ 3 - te (te) te - })
        ({ 3 (te) - te te - })
        ({ 3 (te) (te) te })))
     '((((q q) ; the 2/4 bars: 5 total
        ((q) q)
        ((q) q)
        ((q) (s) e.)
        (- e e - (e) e))
       (({ 3 te+te te+te te+te }) ; what we transition to
        (q - s e. -)
        (q (s) e.)
        (q (s) - s e -)
        ({ 3 te+te te+te - te te - })))
      ((((e.) s (e) e (s) e.) ; the 3/4 bars: 4 total
        (- e e - (e) e (q))
        (- e. s - - +e e - (q))
        (q (e.) s (q)))
       (({ 3 (te) (te) te+te te+te } (q)) ; what we transition to
        (- e. s - (q) (s) - s e -)
        ({ 3 te+te te } (q) q)
        ({ 3 - te te te - } (e) e { 3 (te) (te) te }))))
        :players '(fl cl))))
  (add-voice rch '(1 cl) 'ob))
```

SYNOPSIS:

```
(defmethod add-voice ((rc rthm-chain) parent new-player &optional (offset 1))
```

28.5.396   rthm-chain/hash-least-used

[rthm-chain] [Functions]

DESCRIPTION:

Return the least used key in a hash table. This may be used to retrieve the
number of times the keys have been used as indices, for example.

ARGUMENTS:

  - A hash table. This must be a lisp hash table object whose keys and values
    are all numbers.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start. The lowest key value we'll test. Default = 0.
 - :end. The highest key value we'll test. Default = number of items in the
   hash table.
 - :ignore. A list of keys to ignore when processing. NIL = process all
   keys. Default = NIL.
 - :auto-inc. T or NIL to determine whether the function will automatically
   increment the count of the returned key. T = automatically increment.
   Default = T.
 - :invert. T or NIL to invert the functionality i.e. to return the most
   used rather than the least. Default = NIL.

RETURN VALUE:

 The key of the least used element in the hash table.

EXAMPLE:

```
(let ((h (make-hash-table)))
  (loop for i below 100 do
       (setf (gethash i h) 10000))
  (setf (gethash 10 h) 5
        (gethash 11 h) 4
        (gethash 12 h) 3
        (gethash 13 h) 2)
  (print (hash-least-used h :auto-inc nil))
  (print (hash-least-used h :auto-inc t))
  (print (hash-least-used h :auto-inc t))
  (print (hash-least-used h :auto-inc nil :start 12))
  (setf (gethash 2 h) 0)
  (print (hash-least-used h :auto-inc nil :start 3 :end 11))
  (print (hash-least-used h :auto-inc nil :end 11))
  (print (hash-least-used h :auto-inc nil :ignore '(2))))

=>
13
13
12
13
11
```

2
13

SYNOPSIS:

```
(defun hash-least-used (hash &key (start 0) end ignore (auto-inc t)
                             invert)
```

28.5.397 rthm-chain/hash-least-useds

[rthm-chain] [Functions]

DATE:

 Mary 26th 2016, Edinburgh

DESCRIPTION:

 Same as hash-least-used except it will return the <num> least used items (3
 by default).

SYNOPSIS:

```
(defun hash-least-useds (hash &key (start 0) end ignore (auto-inc t)
                              invert (num 3))
```

28.5.398 rthm-chain/make-rthm-chain

[rthm-chain] [Functions]

DESCRIPTION:

 Create an instance of a rthm-chain object. The rthm-chain class enables the
 algorithmic generation of a rthm-seq-map (with just one section) and its
 associated rthm-seq-palette, which consists in turn of algorithmically
 generated rthm-seq objects.

 The rhythm-seq objects are made up of both faster material based on 1-beat
 groups and slower-moving counterpoint based on 2- or 3-beat groups.

 The rthm-chain class also allows for control of the degree of activity in
 the parts over time through user-specified envelopes.

 Rests are automatically inserted at regular but changing intervals.

Specified 'sticking points' cause individual rhythms to be repeated a
certain number of times. Sticking happens after rests and can also be
controlled with an activity envelope.

NB: Because this method uses the procession method internally, each
    collection of 1-beat-rthms and slower-rthms defined must contain at
    least four items.

NB: Since this method automatically inserts rests into the chains, the user
    may like to implement the various tie-over-rests post-generation
    editing methods. If this is done, the handle-ties method may also be
    recommended, as the tie-over-rests methods only affect printed output
    and not MIDI output.

ARGUMENTS:

- A number, symbol, or string that is to be the ID of the new rthm-chain
  object.
- An integer that is the number of beats to be generated prior to adding
  additional material created from sticking points and the automatic
  addition of rests. For generating a whole piece this will generally be
  in the hundreds, not dozens. Lower numbers might create very limited
  results or make the use of the add-voice method next to useless.
- A list with sublists of rhythms that are to be the 1-beat rhythms used to
  construct the faster-moving material of the rthm-seq-palette. Each
  sublist represents the repertoire of rhythms that will be used by the
  procession method. Each sublist must contain the same number of rthms but
  their number and the number of sublists is open. A transition will be
  made from one group of rhythms to the next over the whole output
  (i.e. not one unit to another within e.g. the 1-beat rhythms) according
  to a fibonacci-transition method. NB Here and below, at least two lists
  are required: what we start with, and what we transition to. If no
  transition is required you could of course just duplicate the rhythms via
  a let variable: (list rhythms rhythms).
- A list with sublists of 2-beat and 3-beat full bars of rhythms used to
  construct the slower-moving counterpoint material of the
  rthm-seq-palette. This will be turned into a rthm-chain-slow object, and
  will therefore remain as lists of unparsed rhythms. Each sublist must
  contain the same number of rthms but their number and the number of
  sublists is open. A transition will be made from one group of rhythms to
  the next over the whole output (i.e. not one unit to another within
  e.g. the 1-beat rhythms) according to a fibonacci-transition method.
  NB: The rhythm units of slower-rthms must be expressed in single beats;
  e.g., a 2/4 bar must consist of q+q rather than h. The consolidate-notes
  method can be called afterwards if desired.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :players. A list of two player IDs. When used in conjunction with a
  slippery-chicken object (which is the standard usage), these must be IDs
  as they are defined in that object's ENSEMBLE slot. The first player will
  play the 1-beat rhythms, the second the slower rhythms.
  Default = '(player1 player2).
- :section-id. An integer that will be used as the ID of the rthm-seq-map
  created. NB: rthm-chain only creates rthm-seq-maps with one section,
  making it possible to create several different rthm-seq-map objects for
  different sections in the given piece, and requiring that these be
  manually assigned IDs. Additionally, any ID given here must match an
  existing ID within the other maps. Default = 1.
- :activity-curve. A list of break-point pairs with y values from 1 to 10
  indicating the amount of activity there should be over the course of the
  piece. A value of 1 indicates that only 1 in 10 beats will have notes
  in/on them, and a value of 10 indicates that all beats will have
  notes. This process uses the patterns given in
  activity-levels::initialize-instance, where 1 means 'play' and 0 means
  'rest'. There are three templates for each level, so that if the curve
  remains on one level of activity for some time it won't always return the
  same pattern; these will be rotated instead. If the activity curve
  indicates a rest for one of the slower-rhythms groups, the whole 2-3 beat
  group is omitted. Default = '(0 10 100 10).
- :do-rests. T or NIL to indicate whether to apply the automatic
  rest-insertion algorithm. T = use. Default = T.
- :rests. A list of rhythmic duration units from which the durations will
  be drawn when using the automatic rest-insertion algorithm. The specified
  rests are used in a sequence determined by a recurring-event
  object. Default = '(e q q. w). NB: Each of these values must not resolve
  to less than one-quarter of the beat basis, either alone or in
  combination, as this could result in an attempt to create meters from
  fractional beats (e.g. 3.25). An error message will be printed in such
  cases.
- :rest-cycle. A list of 2-item lists that indicate the pattern by which
  rests of specific rhythmic durations will be selected from the RESTS slot
  for automatic insertion. The first number of each pair is a 0-based
  position referring to the list of rests in the RESTS slot, and the second
  number is the number of times the rest at that particular position should
  be inserted. (This number does not mean that the selected rest will be
  inserted that many times at once, but rather that each consecutive time
  the rest algorithm selects one rest to be inserted, it will insert that
  specific rest, for the specified number of consecutive times.) For
  example, (0 3) indicates that for the next three times that the rest
  algorithm selects one rest to insert, it will select the rest located at
```

position 0 in the list of rests in the RESTS slot (e by default).
Default ='((0 3) (1 1) (0 2) (2 1) (1 1) (3 1)).
- :rest-re. A list of 2-item lists that indicate the pattern by which rests
  will be automatically inserted. The first number of each pair determines
  how many events occur before inserting a rest, and the second number of
  each pair determines how many times that period will be repeated. For
  example, (2 3) indicates that a rest will be inserted every two events,
  three times in a row. The list passed here will be treated as data for a
  recurring-event object that will be repeatedly cycled through.
  Default = '((2 3) (3 2) (2 2) (5 1) (3 3) (8 1)).
- :do-rests-curve. A list of break-point pairs with y values of either 0 or
  1 indicating whether the do-rests algorithm is active or disabled. These
  values are interpolated between each pair, with all values 0.5 and higher
  being rounded up to 1 and all below 0.5 rounded to 0. Default = NIL.
- :do-sticking. T or NIL to indicate whether the method should apply the
  sticking algorithm. T = apply. Default = T.
- :sticking-rthms. A list of rhythmic units that will serve as the rhythms
  employed by the sticking algorithm. These are generated at initialization
  if not specified here. NB: This list is used to create a list using the
  procession algorithm at initialization, so it is best to apply something
  similar to the default if not accepting the default. If a circular-sclist
  object is provided here, it will be used instead of the default
  procession. Default = '(e e e. q e s).
- :sticking-repeats. A list of integers to indicate the number of
  repetitions applied in sticking segments. When the values of this list
  have been exhausted, the method cycles to the beginning and continues
  drawing from the head of the list again. NB: This list is made into a
  circular-sclist object when the given rthm-chain object is initialized
  unless a circular-sclist object is explicitly provided.
  Default = '(3 5 3 5 8 13 21).
- :sticking-curve. A list of break-point pairs that acts as an activity
  envelope to control the sticking, which always occurs after rests. As
  with the activity curve, this curve can take y values up to 10, but also
  allows 0. A y value of 0 or 1 here refers to either a specific number of
  repeats (1) or none (0). The number of repeats may be determined, for
  example, by use of the procession method, such as
  (procession 34 '(2 3 5 8 13) :peak 1 :expt 3). Every sticking point is
  accompanied by a slower group, which is simply chosen in sequence and
  repeated for the duration of the sticking period.
  Default = '(0 2 100 2).
- :do-sticking-curve. A list of break-point pairs that can be used,
  alternatively, to control whether the sticking algorithm is being applied
  or not at any given point over the course of the piece. The y values for
  this curve should be between 0 and 1, and the decimal fractions achieved
  from interpolation will be rounded. The 1 values resulting from this
  curve will only be actively applied to if do-sticking is set to T.

```
    Default = NIL.
 - :harmonic-rthm-curve. A list of break-point pairs that indicates how many
   slower-rthms will be combined into one rthm-seq (each rthm-seq has a
   single harmony). The default is 2 bars (slower-rthms) per rthm-seq,
   i.e. '(0 2 100 2).
 - :split-data. NIL or a two-item list of integers that are the minimum and
   maximum beat duration of bars generated. If NIL, the bars will not be
   split. These values are targets only; the method may create bars of
   different lengths if the data generated cannot be otherwise split.
   NB: The values given here will apply to a different beat basis depending
   on time signature of each individual bar, rather than on a consistent
   beat basis, such as quarters or eighths. Since this method produces bars
   of different lengths with time signatures of differing beat bases
   (e.g. 16, 8, 4 etc.) before it applies the split algorithm, a minimum
   value of 4, for example, can result in bars of 4/16, 4/8, 4/4 etc.
   Default = '(2 5)
 - :1-beat-fibonacci. T or NIL to indicate whether the sequence of 1-beat
   rhythms is to be generated using the fibonacci-transitions method or the
   processions method. T = use fibonacci-transitions method. Default = NIL.
 - :slow-fibonacci. T or NIL to indicate whether the sequence of the slow
   rhythms will be generated using the fibonacci-transitions method or the
   processions method. This affects the order in which each 2- or 3-beat
   unit is used when necessary, not the order in which each 2- or 3-beat
   unit is selected; the latter is decided by the next element in the DATA
   slot of the rthm-chain-slow object, which simply cycles through
   '(2 3 2 2 3 2 2 3 3 3). T = use fibonacci-transitions method.
   Default = NIL.
```

RETURN VALUE:

 A rthm-chain object.

EXAMPLE:

```
;; An example using a number of the keyword arguments.
(make-rthm-chain
 'test-rch 23
 '((((e) e) ; 4 in total
    (- s (s) (s) s -)
    ({ 3 (te) - te te - })
    ((e.) s))
   (({ 3 (te) te (te) }) ; what we transition to
    ({ 3 - te (te) te - })
    ({ 3 (te) - te te - })
    ({ 3 (te) (te) te })))
 '((((q q) ; the 2/4 bars: 5 total
```

```
     ((q) q)
     ((q) q)
     ((q) (s) e.)
     (- e e - (e) e))
    (({ 3 te+te te+te te+te }) ; what we transition to
     (q - s e. -)
     (q (s) e.)
     (q (s) - s e -)
     ({ 3 te+te te+te - te te - })))
   (((((e.) s (e) e (s) e.) ; the 3/4 bars: 4 total
     (- e e - (e) e (q))
     (- e. s - - +e e - (q))
     (q (e.) s (q)))
    (({ 3 (te) (te) te+te te+te } (q)) ; what we transition to
     (- e. s - (q) (s) - s e -)
     ({ 3 te+te te } (q) q)
     ({ 3 - te te te - } (e) e { 3 (te) (te) te })))))
 :players '(fl cl)
 :slow-fibonacci t
 :activity-curve '(0 1 100 10)
 :harmonic-rthm-curve '(0 1 100 3)
 :do-sticking t
 :do-sticking-curve '(0 1 25 0 50 1 75 0 100 1)
 :sticking-curve '(0 0 100 10)
 :sticking-repeats '(3 5 7 11 2 7 5 3 13)
 :sticking-rthms '(e s. 32 e.)
 :split-data '(4 7))

=>
RTHM-CHAIN: 1-beat-rthms: (((E E) (S S S S) (TE TE TE) (E. S))
                           ((TE TE TE) (TE TE TE) (TE TE TE) (TE TE TE)))
            slower-rthms: ((((Q Q) ((Q) Q) ((Q) Q) ((Q) (S) E.)
                              (- E E - (E) E))
                             (({ 3 TE+TE TE+TE TE+TE }) (Q - S E. -) (Q (S) E.)
                              (Q (S) - S E -) ({ 3 TE+TE TE+TE - TE TE - })))
                            (((((E.) S (E) E (S) E.) (- E E - (E) E (Q))
                              (- E. S - - +E E - (Q)) (Q (E.) S (Q)))
                             (({ 3 (TE) (TE) TE+TE TE+TE } (Q))
                              (- E. S - (Q) (S) - S E -) ({ 3 TE+TE TE } (Q) Q)
                              ({ 3 - TE TE TE - } (E) E { 3 (TE) (TE) TE })))))
            1-beat-fibonacci: NIL
            num-beats: 23
            slow-fibonacci: T
            num-1-beat-rthms: 4
            num-1-beat-groups: 2
            sticking-curve: (0.0 0 22 10)
```

```
            harmonic-rthm-curve: (0.0 1 22 3)
            beat: 4
            do-sticking: T
            do-rests: T
            do-sticking-curve: (0.0 1 5.5 0 11.0 1 16.5 0 22 1)
            do-rests-curve: NIL
            sticking-al: (not printed for brevity's sake)
            sticking-rthms: (E S. E S. 32 E 32 E E E. S. 32 S. E. S. 32 S. 32
                              E. E)
            sticking-repeats: (3 5 3 5 7 3 7 3 3 11 5 7 5 11 5 7 5 7 11 3 7 3 3
                                11 5 7 5 11 3 7 3 7 11 5 7 5 5 11 3 11 3 2 11 2
                                11 2 7 2 7 2 2 5 5 3 5)
            activity-curve: (0.0 1 22 10)
            main-al: (not printed for brevity's sake)
            slower-al: (not printed for brevity's sake)
            num-slower-bars: 22
            rcs: (not printed for brevity's sake)
            rests: (E Q Q. W)
            rest-re: (not printed for brevity's sake)
            rest-cycle: ((0 3) (1 1) (0 2) (2 1) (1 1) (3 1))
            num-rthm-seqs: 19
            section-id: 1
            split-data: (4 7)
RTHM-SEQ-MAP: num-players: 2
              players: (FL CL)
SC-MAP: palette id: RTHM-CHAIN-RSP
[...]
```

SYNOPSIS:

```
(defun make-rthm-chain (id num-beats 1-beat-rthms slower-rthms &key
                        (1-beat-fibonacci nil)
                        (slow-fibonacci nil)
                        (players '(player1 player2))
                        (section-id 1)
                        (rests '(e q q. w))
                        (do-rests t)
                        (do-rests-curve nil)
                        (rest-re '((2 3) (3 2) (2 2) (5 1) (3 3) (8 1)))
                        (rest-cycle '((0 3) (1 1) (0 2) (2 1) (1 1) (3 1)))
                        (activity-curve '(0 10 100 10))
                        (sticking-curve '(0 2 100 2))
                        (harmonic-rthm-curve '(0 2 100 2))
                        (do-sticking t)
                        (do-sticking-curve nil)
                        (sticking-repeats '(3 5 3 5 8 13 21))
```

```
(sticking-rthms '(e e e. q e s))
(split-data '(2 5)))
```

28.5.399   rthm-chain/procession

[rthm-chain] [Functions]

DATE:

26-Jan-2010

DESCRIPTION:

Generate a list of a specified length consisting of items extrapolated from
a specified starting list. All elements of the resulting list will be
members of the original list.

The method generates the new list by starting with the first 3 elements of
the initial list and successively adding consecutive elements from the
initial list until all elements have been added.

ARGUMENTS:

- An integer that is the number of items in the list to be generated.
- A list of at least 4 starting items or an integer >=4. If an integer is
  given rather than a list, the method will process a list of <integer>
  consecutive numbers from 1 (by default).

OPTIONAL ARGUMENTS:

keyword arguments:
- :peak. A decimal number >0.0 and <=1.0. This number indicates the target
  location in the new list at which the last element is to finally occur,
  whereby e.g. 0.7 = ~70% of the way through the resulting list. This is an
  approximate value only. The last element may occur earlier or later
  depending on the values of the other arguments. In particular, initial
  lists with a low number of items are likely to result in new lists in
  which the final element occurs quite early on, perhaps even nowhere near
  the specified peak value. Default = 0.7.
- :expt. An exponent (floating point number) to indicate the "curve" that
  determines the intervals at which each successive element of the initial
  list is introduced to the new list. A higher number indicates a steeper
  exponential curve. Default = 1.3.
- :start-at. The number to start counting from if the 2nd argument is an
  integer instead of a list. Default = 1.

 – :orders. The patterns by which the elements are added. The method
   cyclically applies these orders, the numbers 1, 2, and 3 representing the
   three least used elements at each pass. These orders must therefore
   contain all of the numbers 1, 2, and 3, and those numbers only.
   Default = '((1 2 1 2 3) (1 2 1 1 3) (1 2 1 3)).

RETURN VALUE:

 Returns two values, the first being the new list, with a secondary value
 that is a list of 2-item lists that show the distribution of each element
 in the new list.

EXAMPLE:

(procession 300 30 :peak 0.1)

=>
(1 2 1 2 3 4 5 4 4 6 7 8 7 9 10 11 10 11 12 13 14 13 13 15 16 17 16 18 19 20 19
    20 21 22 23 22 22 24 25 26 25 27 28 29 28 29 30 3 5 3 3 6 8 9 8 12 14 15 14
    15 17 18 21 18 18 23 24 26 24 27 1 2 1 2 30 5 6 5 5 7 9 10 9 11 12 16 12 16
    17 19 20 19 19 21 23 25 23 26 27 28 27 28 29 4 6 4 4 30 7 8 7 10 11 13 11 13
    14 15 17 15 15 20 21 22 21 24 25 26 25 26 29 1 2 1 1 30 3 6 3 8 9 10 9 10 12
    14 16 14 14 17 18 20 18 22 23 24 23 24 27 28 29 28 28 30 2 5 2 6 7 8 7 8 11
    12 13 12 12 16 17 19 17 20 21 22 21 22 25 26 27 26 26 29 3 4 3 30 5 6 5 6 9
    10 11 10 10 13 15 16 15 18 19 20 19 20 23 24 25 24 24 27 1 29 1 30 2 4 2 4 7
    8 9 8 8 11 13 14 13 16 17 18 17 18 21 22 23 22 22 25 27 28 27 29 3 5 3 5 30
    6 7 6 6 9 11 12 11 14 15 16 15 16 19 20 21 20 20 23 25 26 25 28 1 29 1 29 30
    2 4 2 2 7 9 10 9 12 13 14 13 14 17 18), ((2 12) (20 11) (14 11) (13 11)
    (9 11) (6 11) (1 11) (29 10) (25 10) (22 10) (18 10) (17 10) (16 10) (15 10)
    (12 10) (11 10) (10 10) (8 10) (7 10) (5 10) (4 10) (3 10) (30 9) (28 9)
    (27 9) (26 9) (24 9) (23 9) (21 9) (19 9))

(procession 300 30 :peak 0.9)

=>
(1 2 1 2 3 1 3 1 1 4 2 3 2 4 3 4 3 4 5 2 4 2 2 5 1 3 1 5 3 4 3 4 5 1 5 1 1 6 2
    5 2 6 4 5 4 5 6 3 6 3 3 7 5 6 5 7 2 6 2 6 7 6 7 6 6 8 4 7 4 8 7 8 7 8 9 7 8
    7 7 9 8 9 8 10 8 9 8 9 10 8 9 8 8 10 9 10 9 11 9 10 9 10 11 10 11 10 10 12
    10 11 10 12 11 12 11 12 13 11 12 11 11 13 12 13 12 14 12 13 12 13 14 13 14
    13 13 15 13 14 13 15 11 14 11 14 15 14 15 14 14 16 15 16 15 17 15 16 15 16
    17 16 17 16 16 18 16 17 16 18 17 18 17 18 19 17 18 17 17 19 18 19 18 20 18
    19 18 19 20 19 20 19 19 21 15 20 15 21 20 21 20 21 22 20 21 20 20 22 21 22
    21 23 21 22 21 22 23 22 23 22 22 24 23 24 23 25 23 24 23 24 25 24 25 24 24
    26 23 25 23 26 25 26 25 26 27 26 27 26 26 28 25 27 25 28 27 28 27 28 29 27
    28 27 27 29 28 29 28 30 24 29 24 29 30 26 29 26 26 30 28 29 28 30 19 29 19
    29 30 22 25 22 22 30 12 27 12 30 14 16 14 16 30 17), ((8 12) (22 11)

```
   (16 11) (14 11) (12 11) (11 11) (10 11) (4 11) (3 11) (2 11) (26 10)
   (19 10) (17 10) (15 10) (13 10) (9 10) (7 10) (6 10) (5 10) (1 10)
   (29 9) (28 9) (27 9) (25 9) (24 9) (23 9) (21 9) (20 9) (18 9) (30 8))
```

SYNOPSIS:

```
(defun procession (num-results items
                    &key
                        ;; what proportion of the way through should we aim to
                        ;; reach the max number of items?  NB This is approximate
                        ;; only: you may find the first occurrence of the highest
                        ;; element earlier or later depending on the values of the
                        ;; other arguments.  In particular, with a low number of
                        ;; items the highest element will be hit very early on,
                        ;; perhaps nowhere near the peak argument.
                        (peak 0.7)
                        ;; for an exponential curve going from 3 to num <items>
                        (expt 1.3)
                        ;; MDE Thu Sep 17 16:11:03 2020, Heidhausen -- allow
                        ;; counting from another number if items is an integer
                        (start-at 1)
                        ;; these are the orders we'll use at the beginning
                        ;; (cyclically). They will then be used when we've gone
                        ;; beyond 3 items by always using the 3 least used items.
                        ;; NB This must contain the numbers 1, 2, and 3 only but
                        ;; there can be 1 or any number of sublists.
                        (orders '((1 2 1 2 3) (1 2 1 1 3) (1 2 1 3)))))
```

28.5.400 rthm-chain/procession-mirror

[rthm-chain] [Functions]

DESCRIPTION:

 Perform the same operation as the procession function, but instead of just
 progressing upwards, go backwards to return to the beginning also.

ARGUMENTS:

 The same as for procession.

RETURN VALUE:

 A list of the procession. Note that at the mirror point there is no
 repetition, and we don't include the first element at the end. Note also
 that we don't return statistics as we do with procession.

EXAMPLE:

```
(procession-mirror 33 '(1 2 3 4 5 6 7))
--> (1 2 1 2 3 3 4 3 3 5 4 6 4 7 5 6 5 6 6 5 7 4 6 4 5 3 3 4 3 3 2 1 2)
```

SYNOPSIS:

```
(defun procession-mirror (&rest args)
```

28.5.401   rthm-chain/reset

[rthm-chain] [Methods]

DESCRIPTION:

 Reset the various circular-sclist objects within the given rthm-chain
 object to their initial state.

ARGUMENTS:

 - A rthm-chain object.

OPTIONAL ARGUMENTS:

 (- :where. This argument is ignored by the method as it is only present due
    to inheritance.)

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;;; Print the results of applying get-next to the STICKING-RTHMS slot of the
;;; given rthm-chain object, repeat, reset, and print again to see that the
;;; get-next now begins at the beginning of the slot again.

(let ((rch
       (make-rthm-chain
        'test-rch 150
        '((((e) e) ; 4 in total
           (- s (s) (s) s -)
           ({ 3 (te) - te te - })
           ((e.) s))
```

```
               (({ 3 (te) te (te) }) ; what we transition to
                ({ 3 - te (te) te - })
                ({ 3 (te) - te te - })
                ({ 3 (te) (te) te })))
           '((((q q) ; the 2/4 bars: 5 total
                ((q) q)
                ((q) q)
                ((q) (s) e.)
                (- e e - (e) e))
               (({ 3 te+te te+te te+te }) ; what we transition to
                (q - s e. -)
                (q (s) e.)
                (q (s) - s e -)
                ({ 3 te+te te+te - te te - })))
             ((((e.) s (e) e (s) e.) ; the 3/4 bars: 4 total
                (- e e - (e) e (q))
                (- e. s - - +e e - (q))
                (q (e.) s (q)))
               (({ 3 (te) (te) te+te te+te } (q)) ; what we transition to
                (- e. s - (q) (s) - s e -)
                ({ 3 te+te te } (q) q)
                ({ 3 - te te te - } (e) e { 3 (te) (te) te })))))))
  (print
   (loop repeat 19
      collect (data (get-next (sticking-rthms rch)))))
  (print
   (loop repeat 19
      collect (data (get-next (sticking-rthms rch)))))
  (reset rch)
  (print
   (loop repeat 19
      collect (data (get-next (sticking-rthms rch))))))

=>
(E E E E E. E E. E E Q E E. E Q E. Q E. Q E)
(E E E E E E. E E. E E Q E E. E Q E. Q E. Q)
(E E E E E. E E. E E Q E E. E Q E. Q E. Q E)
```

SYNOPSIS:

```
(defmethod reset ((rc rthm-chain) &optional ignore1 ignore2)
```

28.5.402   rthm-chain/rthm-chain-gen

[rthm-chain] [Methods]

DESCRIPTION:

Generate a chain of rhythms using the procession function (internally).

The basic algorithm for generating a rthm-chain object of two parts is as
follows: The user provides an arbitrary number of 1-beat rthms (e.g. s s
(e)) and 2-3 beat slower-moving counterpoints. The method generates a
sequence from these using the procession function. Next the activity curve
is applied to this, and after that the insertion of rests. Then the
'sticking points' are generated: These come after the rests, and the
activity curves applied to these count inserted rests not seqs or beats.

NB: Rests are put into the given rthm-seq object mid-sequence, so sticking
    points won't come directly after the rests, rather, at the end of the
    seq.

The activity curves that turn notes into rests will be queried every beat,
so if an activity level is changed, the method won't wait until the end of
the previous level's ten beats.

NB: This method is not generally called by the user (though it can be of
    course); rather, it's called by the init function.

ARGUMENTS:

 - A rthm-chain object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :rests. T or NIL to indicate whether rests are to be automatically
  inserted. T = automatically insert. Default = T.
- :stick. T or NIL to indicate whether to generate the sticking points. T =
  generate sticking points. Default = T.
- :num-beats. NIL or an integer to indicate how many beats are to be used
  for the algorithm. NB: The method will generate considerably more beats
  if also generating sticking points and inserting rests; this number
  merely refers to the number of standard 1-beat rhythms to be generated.
  If NIL, the method will obtain the number of beats from the NUM-BEATS
  slot of the rthm-chain instance. Default = NIL.
- :use-fibonacci. T or NIL to indicate whether to use the
  fibonacci-transitions method when generating the sequence from the 1-beat
  rhythms (in which case these will be repeated) or the procession
  algorithm (in which case they'll be alternated). T = use the
  fibonacci-transitions method. Default = T.

- :section-id. An integer that is the section ID of the rthm-chain object
  to be generated. This will determine the section of the rthm-seq-map into
  which the references will be placed. The rthm-seq objects themselves will
  also be parcelled up into an object with this ID, so ID conflicts can be
  avoided if combining two or more sections generated by separate
  rthm-chain objects. Default = 1.
- :wrap. An integer or NIL to determine the position within the list of
  1-beat rhythms and slow rhythms from which the generated rhythm chain
  will begin. NIL = begin at the beginning. Default = NIL.
- :split. T or NIL to indicate whether to split up longer generated bars
  (e.g. 7/4) into smaller bars. If this is a two-element list it represents
  the min/max number of beats in a bar (where a 6/8 bar is two compound
  beats). Default = '(2 5).

RETURN VALUE:

 the number of rthm-seqs we've generated

SYNOPSIS:

```
(defmethod rthm-chain-gen ((rc rthm-chain)
                           &key
                           (use-fibonacci t)
                           (rests t)
                           (stick t)
                           (section-id 1)
                           num-beats
                           wrap)
```

28.5.403   rthm-chain/split

[rthm-chain] [Methods]

DATE:

 29-Jan-2011

DESCRIPTION:

 Split the longer generated bars into smaller ones where possible.

ARGUMENTS:

 - A rthm-chain object.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :min-beats. An integer that is the minimum number of beats in the
  resulting bars. This is a target-length only, and may not be adhered to
  strictly if durations do not allow. Default = 2.
- :max-beats. An integer that is the maximum number of beats in the
  resulting bars. This is a target-length only, and may not be adhered to
  strictly if durations do not allow. Default = 5.
- :warn. T or NIL to indicate whether to print a warning to the listener if
  the current bar cannot be split. T = print. Default = NIL.
- :clone. T or NIL to indicate whether the rthm-seq of the given rthm-chain
  object should be changed in place or changes should be made to a copy of
  that object. T = create a copy to be changed. Default = T.
```

RETURN VALUE:

```
 Returns T.
```

EXAMPLE:

```
;;; Make a rthm-chain object using make-rthm-chain with the :split-data
;;; argument set to NIL and print the number of bars in each resulting rthm-seq
;;; object. Apply the split method and print the number of bars again to see
;;; the change.

(let* ((rch
        (make-rthm-chain
         'test-rch 150
         '((((e) e) ; 4 in total
            (- s (s) (s) s -)
            ({ 3 (te) - te te - })
            ((e.) s))
           (({ 3 (te) te (te) }) ; what we transition to
            ({ 3 - te (te) te - })
            ({ 3 (te) - te te - })
            ({ 3 (te) (te) te })))
         '((((q q) ; the 2/4 bars: 5 total
             ((q) q)
             ((q) q)
             ((q) (s) e.)
             (- e e - (e) e))
            (({ 3 te+te te+te te+te }) ; what we transition to
             (q - s e. -)
             (q (s) e.)
```

```
            (q (s) - s e -)
            ({ 3 te+te te+te - te te - })))
          ((((e.) s (e) e (s) e.) ; the 3/4 bars: 4 total
            (- e e - (e) e (q))
            (- e. s - - +e e - (q))
            (q (e.) s (q)))
           (({ 3 (te) (te) te+te te+te } (q)) ; what we transition to
            (- e. s - (q) (s) - s e -)
            ({ 3 te+te te } (q) q)
            ({ 3 - te te te - } (e) e { 3 (te) (te) te }))))
          :split-data nil)))
  (print
   (loop for rs in (data (get-data-data 1 (palette rch)))
      collect (num-bars rs)))
  (split rch :min-beats 1 :max-beats 3 :clone nil)
  (print
   (loop for rs in (data (get-data-data 1 (palette rch)))
      collect (num-bars rs))))

=>
(1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 1 1 2 2 1 1 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2
 2 2)
(1 1 4 4 2 7 4 4 3 3 4 4 2 9 7 7 2 5 5 5 2 6 1 1 2 13 3 3 4 4 5 5 2 7 5 5
 7 7 9 9 2 7 2 9 3 3 2 9 1 1 5 5 9 9 3 3 2 7 5 5 4 4 2 11 1 1 2 10 2 9 2 6
 7 7 7 7)
```

SYNOPSIS:

```
(defmethod split ((rc rthm-chain) &key
                                  (min-beats 2) (max-beats 5) warn (clone t))
```

28.5.404   rthm-seq-map/set-map-refs

[rthm-seq-map] [Methods]

DATE:

 30-Dec-2010

DESCRIPTION:

 Change the reference IDs of the specified rthm-seq objects in the given
 rthm-seq-map object.

ARGUMENTS:

- A rthm-seq-map object.
- The ID of the section in which references are to be set.
- The ID of the player for whom the references are to be set.
- A list of the new rthm-seq IDs (references)

RETURN VALUE:

Returns the modified named object whose ID is the specified player.

EXAMPLE:

```
(let ((rsmt (make-rthm-seq-map
             'rsm-test-5
             '((sec1 ((vn (rs1 rs3 rs2))
                      (va (rs2 rs3 rs1))
                      (vc (rs3 rs1 rs2))))
               (sec2 ((vn (rs1 rs2 rs1))
                      (va (rs2 rs1 rs3))
                      (vc (rs1 rs3 rs3))))
               (sec3 ((vn (rs1 rs1 rs3))
                      (va (rs1 rs3 rs2))
                      (vc (rs3 rs2 rs3)))))
             :palette (make-rsp
                       'rs-pal
                       '((rs1 ((((2 4) q e s s))))
                         (rs2 ((((2 4) e s s q))))
                         (rs3 ((((2 4) s s q e)))))))))
  (set-map-refs rsmt 'sec2 'vc '(rs2 rs3 rs2)))

=>
NAMED-OBJECT: id: VC, tag: NIL,
data: (RS2 RS3 RS2)
```

SYNOPSIS:

```
(defmethod set-map-refs ((rsm rthm-seq-map) section player new-refs)
```

28.5.405   sc-map/set-map

[sc-map] [Classes]

NAME:

set-map

```
File:               set-map.lsp

Class Hierarchy:    named-object -> linked-named-object -> sclist ->
                    circular-sclist -> assoc-list -> recursive-assoc-list ->
                    sc-map -> set-map

Version:            1.0.11

Project:            slippery chicken (algorithmic composition)

Purpose:            Implementation of the set-map class for mapping sets for
                    a piece.

Author:             Michael Edwards: m@michael-edwards.org

Creation date:      March 11th 2010

$$ Last modified:  18:42:21 Fri May  1 2020 CEST

SVN ID: $Id$
```

28.5.406   set-map/cmn-display

[set-map] [Methods]

DATE:

 January 30th 2016

DESCRIPTION:

 Generate musical notation of the set-map object using CMN.

ARGUMENTS:

 a set-map object

OPTIONAL ARGUMENTS:

 see the set-palette class for a description of the keyword arguments.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod cmn-display ((sm set-map) &rest keyargs &key &allow-other-keys)
```

28.5.407   set-map/gen-midi-chord-seq

[set-map] [Methods]

DESCRIPTION:

 Write a midi file with each set in the set-map played as a chord at 1
 second intervals.

ARGUMENTS:

 - A set-map object.
 - The path+file-name for the midi file to be written.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((c3 e3 g3 a3 c4 d4 g4 a4 b4 e5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))
        :rthm-seq-map '((1 ((vn (1 1 1))))))))
  (gen-midi-chord-seq (set-map mini) "/tmp/mchsq.mid"))

=> T
```

SYNOPSIS:

```
(defmethod gen-midi-chord-seq ((sm set-map) midi-file &optional (tempo 60.0))
```

28.5.408   sc-map/shorten

[sc-map] [Methods]

DATE:

July 10th 2020, Heidhausen

DESCRIPTION:

 shorten the length of the map for a given section

ARGUMENTS:

 - the section reference
 - the new length (integer)
 - the sc-map object

RETURN VALUE:

 the shortened sc-map object

SYNOPSIS:

(defmethod shorten (section new-length (scm sc-map))

28.5.409   recursive-assoc-list/set-data

[recursive-assoc-list] [Methods]

DESCRIPTION:

 Replace the named-object associated with a specified key within a given
 recursive-assoc-list object. This method replaces the whole named-object,
 not just the data of that object.

ARGUMENTS:

 - A key present within the given recursive-assoc-list object. This must be
   a list that is the FULL-REF (path of keys) if replacing a nested
   named-object. If replacing a named-object at the top level, the
   key can be given either as a single-item list or an individual symbol.
 - A key/data pair as a list.
 - The recursive-assoc-list object in which to find and replace the
   named-object associated with the specified key.

RETURN VALUE:

 Returns the new named-object.

 Returns NIL when the specified key is not found within the given
 recursive-assoc-list object.

EXAMPLE:

```
;;; Replace a named-object at the top level using a single symbol
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (set-data 'wild '(makers mark) ral))

=>
NAMED-OBJECT: id: MAKERS, tag: NIL,
data: MARK

;; The same can be done stating the top-level key as a single-item list. Apply
;; the get-all-refs method in this example to see the change
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
                                        (white ribbon)))))))))
  (set-data '(wild) '(makers mark) ral)
  (get-all-refs ral))

=> ((JIM) (MAKERS) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
    (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED FOX) (FOUR VIOLETS WHITE))

;; Replace a nested named-object using a list that is the FULL-REF to that
;; object. Print the application of the method as well as the results from
;; applying the get-all-refs method in this example to see the effects
(let ((ral (make-ral 'mixed-bag
                     '((jim beam)
                       (wild turkey)
                       (four ((roses red)
                              (violets ((blue velvet)
                                        (red ((dragon den)
                                              (viper nest)
                                              (fox hole)))
```

```
                                              (white ribbon)))))))))
  (print (set-data '(four violets red fox) '(bee hive) ral))
  (print (get-all-refs ral)))

=>
NAMED-OBJECT: id: BEE, tag: NIL,
data: HIVE
**************

((JIM) (WILD) (FOUR ROSES) (FOUR VIOLETS BLUE) (FOUR VIOLETS RED DRAGON)
 (FOUR VIOLETS RED VIPER) (FOUR VIOLETS RED BEE) (FOUR VIOLETS WHITE))
```

SYNOPSIS:

```
(defmethod set-data (key new-value (ral recursive-assoc-list))
```

28.5.410   recursive-assoc-list/set-slot

[recursive-assoc-list] [Methods]

DESCRIPTION:

Set the specified slot of an object with a recursive-assoc-list structure
to the specified value. This is particularly useful for changing the
parameters of instrument objects within an instrument palette, for example.

NB Setting, for instance, the high-sounding slot of an instrument will do
nothing to the highest-written slot so do take care to set all the slots
you need to in order to effect pitch selection etc. Or better still use the
set-standard-range function.

ARGUMENTS:

- The name of the slot whose value is to be set.
- The value to which that slot is to be set.
- The key within the given recursive-assoc-list object for which the slot
  is to be set.
- The recursive-assoc-list object in which the slot is to be changed.

RETURN VALUE:

The value to which the slot has been set.

EXAMPLE:

```
(set-slot 'largest-fast-leap 10 'oboe
         +slippery-chicken-standard-instrument-palette+)
```

```
=> 10
```

SYNOPSIS:

```
(defmethod set-slot (slot value id (ral recursive-assoc-list))
```

28.5.411   recursive-assoc-list/set-standard-instrument-slot

[recursive-assoc-list] [Methods]

DESCRIPTION:

```
 A convenience method to set slots of instruments in the
 +slippery-chicken-standard-instrument-palette+.
```

DATE:

```
 July 20th 2015
```

EXAMPLE:

```
(set-standard-instrument-slot 'highest-written 'f4 'double-bass)
```

SYNOPSIS:

```
(defmethod set-standard-instrument-slot (slot value id)
```

28.5.412   assoc-list/reindex

[assoc-list] [Methods]

DATE:

```
 14th June 2017, Edinburgh
```

DESCRIPTION:

```
 Re-index the named-objects to consecutive integers starting from a given
 number.
```

ARGUMENTS:

- the assoc-list object

OPTIONAL ARGUMENTS:

- the integer to start reindexing at

RETURN VALUE:

 the assoc-list object

SYNOPSIS:

(defmethod reindex ((al assoc-list) &optional (start 1)))

28.5.413   assoc-list/remove-data

[assoc-list] [Methods]

DATE:

 February 1st 2016

DESCRIPTION:

 Convenience function to allow objects to be removed from an assoc-list (and
 subclasses e.g. palettes and maps)

ARGUMENTS:

 - the assoc-list object
 - as many keys as you like for the members of the assoc-list. If none are
   passed then all are removed.

RETURN VALUE:

 The assoc-list object's new data list, i.e. with elements removed. Could of
 course be NIL if you've not passed any keys.

SYNOPSIS:

(defmethod remove-data ((al assoc-list) &rest keys)

28.5.414   assoc-list/remove-when

[assoc-list] [Methods]
DATE:

 June 28th 2017, Edinburgh

DESCRIPTION:

 Use a function to test each object in an assoc-list and if the test returns
 T, then remove the object. Works in recursive-assoc-list class too.

ARGUMENTS:

 - an assoc-list object
 - a function that returns T or NIL after interrogating a named-object or
   subclass's slots

RETURN VALUE:

 a list of the IDs of the objects removed.

EXAMPLE:

```
(let ((al (make-assoc-list 'mixed-bag
                            '((jim beam)
                              (wild turkey)
                              (four roses)))))
  (remove-when al #'(lambda (x) (eq (data x) 'roses))))
```

SYNOPSIS:

```
(defmethod remove-when ((al assoc-list) test)
```

28.5.415   assoc-list/replace-data

[assoc-list] [Methods]
DESCRIPTION:

 A convenience method that's essentially the same as set-data but without
 having to pass the new ID in a list along with the new value.

ARGUMENTS:

  - The key (number, symbol, string) to the existing data.
  - The new value to be associated with this key (any data).
  - The assoc-list object

RETURN VALUE:

 The named-object for the datum.

SYNOPSIS:

(defmethod replace-data (key new-value (al assoc-list))

28.5.416   assoc-list/set-data

[assoc-list] [Methods]

DESCRIPTION:

 Replace the named-object associated with a specified key within a given
 assoc-list object. This method replaces the whole named-object, not just
 the data of that object.

ARGUMENTS:

 - A key present within the given assoc-list object.
 - A key/data pair as a list.
 - The assoc-list object in which to find and replace the named-object
   associated with the specified key.

RETURN VALUE:

 Returns the new named-object.

 Returns NIL when the given key is not present within the given assoc-list.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog fido)
                                   (cow bessie)))))
  (set-data 'dog '(dog spot) al))

=>
NAMED-OBJECT: id: DOG, tag: NIL,
```

```
data: SPOT

(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog fido)
                                   (cow bessie)))))
  (set-data 'pig '(pig wilbur) al))

=> NIL
WARNING:
   assoc-list::set-data: Could not find data with key PIG in assoc-list with id
   TEST

(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog fido)
                                   (cow bessie)))))
  (set-data 'dog '(pig wilbur) al))

=>
NAMED-OBJECT: id: PIG, tag: NIL,
data: WILBUR
```

SYNOPSIS:

```
(defmethod set-data (key new-value (al assoc-list))
```

28.5.417   assoc-list/set-nth-of-data

[assoc-list] [Methods]

DESCRIPTION:

 Replace a given member of a given data list within a given assoc-list.

ARGUMENTS:

 - The key (named-object id) associated with the data to be changed.
 - The zero-based integer index of the member of the list to be changed.
 - The new value.
 - The assoc-list in which the change is to be made.

 The data to be modified must already be in the form of a list.

 The index integer given must be less than the length of the data list to be
 modified.

RETURN VALUE:

```
 Returns the new value only.
```

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog (fido spot rover))
                                   (cow bessie)))))
  (set-nth-of-data 'dog 0 'snoopy al))

=> SNOOPY

(let ((al (make-assoc-list 'test '((cat felix)
                                   (dog (fido spot rover))
                                   (cow bessie)))))
  (set-nth-of-data 'dog 0 'snoopy al)
  (get-data 'dog al))

=>
NAMED-OBJECT: id: DOG, tag: NIL,
data: (SNOOPY SPOT ROVER)
```

SYNOPSIS:

```
(defmethod set-nth-of-data (key nth new-value (al assoc-list))
```

28.5.418   assoc-list/wolfram

[assoc-list] [Classes]

NAME:

```
 rhythm

 File:            wolfram.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> assoc-list -> wolfram

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         Implementation of the wolfram class for one-dimensional
                  binary cellular automata introduced by Stephen Wolfram in
                  1983. The rule number is the decimal representation of
```

> the binary number indicated by the eight transition rules
> (i.e. the results given the triplet test cases; see the
> rules slot below).

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 7th June 2017, Edinburgh

 $$ Last modified: 17:16:57 Mon Feb 10 2020 CET

 SVN ID: $Id: wolfram.lsp 6210 2017-04-07 11:42:29Z medward2 $

28.5.419   wolfram/defwolfram

[wolfram] [Methods]

DATE:

 June 6th 2017, Edinburgh

DESCRIPTION:

 Define your own Wolfram function to create an object with the flavour in
 the rules passed. See below for several examples.

 In addition, the name of the newly created function will be added to an
 internal list of all functions defined by this macro, so that (try-all) can
 be called (see below).

ARGUMENTS:

 - the name of the function you're defining (symbol)
 - the list of rules

OPTIONAL ARGUMENTS:

 - the default width (integer>0) for the Wolfram object that will be created
 - the default initial-state: 1 or 0.

RETURN VALUE:

 the name of the function (symbol), just as with defun.

SYNOPSIS:

(defmacro defwolfram (fun rules &optional (w 100) (is 0))

28.5.420   wolfram/generate

[wolfram] [Methods]

DATE:

 June 6th 2017, Edinburgh

DESCRIPTION:

 Generate the cellular automata rows using the given rules. N.B. Each time
 this method is called, the results (data slot) of the previous calls are
 deleted.

ARGUMENTS:

 - the wolfram object
 - the number of generations (rows) to generate

OPTIONAL ARGUMENTS:

 <start> is the value of the middle cell of the first row (or just to the
 left of middle, if width is an even number). All cells to the left and
 right of it will be in <initial-state>. This should be 0 or 1 only.

RETURN VALUE:

 The data slot, which will be the rows in the form of a list of
 named-objects containing the ID (starting from 1) and the list of cell
 values.

SYNOPSIS:

(defmethod generate ((w wolfram) generations &optional (start 1))

28.5.421   wolfram/make-wolfram-r110

[wolfram] [Functions]

DATE:

 June 7th 2017, Edinburgh.

DESCRIPTION:

```
#b01101110 = 110
```
This is the one Wolfram suggests might be Turing complete. This was proved
in 2000. "Rule 110, like the Game of Life, exhibits what Wolfram calls
"Class 4 behavior", which is neither completely stable nor completely
chaotic. Localized structures appear and interact in various
complicated-looking ways" (https://en.wikipedia.org/wiki/Rule_110)

ARGUMENTS:

none required

OPTIONAL ARGUMENTS:

- the width of a row. Default = 100.
- the initial state of the object. Default = 0.

RETURN VALUE:

the function name as a symbol but when called a wolfram object with the
given rules.

EXAMPLE:

so this one only goes up to column 7 when called with width of 12:
```
     .
     ..
     ...
    .. .
   .....
 ..    .
... ..
. . ..
..... .
.   ...
.  .. .
. .....
...   .
.. . ..
... ...
. ... .
```
then the last 2 lines repeat

but called with width of 60:
```
                         .
                         ..
```

```
.   .  ... ..    ... ... ..
.  .. .. . ...   .. ... . ...
. .......... .   ..... ..... .
. ..       ... ..  ...   ...
....       .. . ... ... . .. .
. .        ....... . ..... .....
. ..     ..    .....   ...   .
|#
(defwolfram make-wolfram-r110
    '(((1 1 1) 0) ((1 1 0) 1) ((1 0 1) 1) ((1 0 0) 0) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.422   wolfram/make-wolfram-r126

[*wolfram*] [*Functions*]

DATE:

 June 6th 2017, Edinburgh.

DESCRIPTION:

 #b01111110 = 126

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

 - the width of a row. Default = 100.
 - the initial state of the object. Default = 0.

RETURN VALUE:

 the function name as a symbol but when called a wolfram object with the
 given rules.

EXAMPLE:

```
width: 12, initial-state: 0
      .
     ...
    .. ..
```

```
    .......
   ..      ..
  ....    ....
 ..  ..  ..  .
 .............
 .           .
 ..         ..
 ...       ...
 .  ..    ..  .
 .....   .....
 .   ....    .
 ..  ..  ..  ..
 .............
using 1, the following repeats but not forever
 .............

 .            .
 ..          ..
  ..        ..
 ....      ....
    ..    ..
 .  ........ .
 ...        ...
   ..      ..
 .....    .....
      ....
 .  ..  ..  .
```

SYNOPSIS:

```
(defwolfram make-wolfram-r126
    '(((1 1 1) 0) ((1 1 0) 1) ((1 0 1) 1) ((1 0 0) 1) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.423   wolfram/make-wolfram-r150

[wolfram] [Functions]

DATE:

 June 7th 2017, Edinburgh.

DESCRIPTION:

 #b10010110 = 150

ARGUMENTS:

```
none required
```

OPTIONAL ARGUMENTS:

```
- the width of a row. Default = 100.
- the initial state of the object. Default = 0.
```

RETURN VALUE:

```
the function name as a symbol but when called a wolfram object with the
given rules.
```

EXAMPLE:

```
width: 12, initial-state: 0
non-repeating  but lovely patterns
      .
      ...
     .  .  .
     ..  .  ..
    .   .   .
   ... ... ...
  .  .   .   .
  .  ..  ...  ...
  .   .   .
  ..   ...  ...
    .  .  .   .
   ..  .  ..  ...
  .   .    .
  ..  ...    ...
      .  .  .  .
      ..  .  .  ..
     .     .  .
   ...  ..  ..
  .  .        .
  .  ..       ...
  .   .    .  .
  ..  ...  ..  ..
       .
       ...
      .  .  .
     ..  .  ..
    .    .    .
```

```
.. ... ...
   .   . .
  ... .. ..
 . .
.. ..
.      .
..    ...
 . . . .
.. .. . ..
.  .  .   .
.. .. .. ...
         .
         ...
        . .
       .. ..
      .
     ...
    . . .
   .. . ..
  .   .   .
 ... ... ...
```

SYNOPSIS:

```
(defwolfram make-wolfram-r150
    '(((1 1 1) 1) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 0)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.424   wolfram/make-wolfram-r151

[wolfram] [Functions]

DATE:

 June 7th 2017, Edinburgh.

DESCRIPTION:

 #b10010111 = 151

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

- the width of a row. Default = 100.
- the initial state of the object. Default = 0.

RETURN VALUE:

the function name as a symbol but when called a wolfram object with the
given rules.

EXAMPLE:

```
         .
...........
 ..........
. ........ .
.  ......  .
... .... ...
 .    ..    .
.....  .....
 ... .. ...
. .        . .
. ........ .
.  ......  .
then the last 7 lines repeat
|#
(defwolfram make-wolfram-r151
   '(((1 1 1) 1) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 0)
     ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 1)))
```

28.5.425   wolfram/make-wolfram-r159

[wolfram] [Functions]
DATE:

June 7th 2017, Edinburgh.

DESCRIPTION:

#b10011111 = 159

ARGUMENTS:

none required

OPTIONAL ARGUMENTS:

- the width of a row. Default = 100.
- the initial state of the object. Default = 0.

RETURN VALUE:

the function name as a symbol but when called a wolfram object with the
given rules.

EXAMPLE:

```
        .
...........
..........
......... .
........  .
.......  ...
......   ..
...... ... .
.....  ..  .
.... ... ...
...  ..  ..
.. ... ... .
.  ..  ..  .
then the last 4 lines repeat
|#
(defwolfram make-wolfram-r159
    '(((1 1 1) 1) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 1)))
```

28.5.426   wolfram/make-wolfram-r190

[wolfram] [Functions]
DATE:

June 6th 2017, Edinburgh.

DESCRIPTION:

#b10111110 = 190

ARGUMENTS:

none required

OPTIONAL ARGUMENTS:

```
- the width of a row. Default = 100.
- the initial state of the object. Default = 0.
```

RETURN VALUE:

```
the function name as a symbol but when called a wolfram object with the
given rules.
```

EXAMPLE:

```
width: 12, initial-state: 0
       .
      ...
      ... .
     ... ...
    ... ... .
   ... ... ...
  ... ... ...
  .. ... ... .
  . ... ... ..
  .... ... ..
  ... ... .. .
  .. ... .. ..
  . ... .. ..
  .... .. .. .
  ... .. .. ..
  .. .. .. ..
  . .. .. .. .
then last three rows repeat
```

SYNOPSIS:

```
(defwolfram make-wolfram-r190
    '(((1 1 1) 1) ((1 1 0) 0) ((1 0 1) 1) ((1 0 0) 1) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.427   wolfram/make-wolfram-r25

[wolfram] [Functions]

DATE:

```
June 6th 2017, Edinburgh.
```

DESCRIPTION:

```
#b00011001 = 25
```

ARGUMENTS:

```
none required
```

OPTIONAL ARGUMENTS:

```
- the width of a row. Default = 100.
- the initial state of the object. Default = 0.
```

RETURN VALUE:

```
the function name as a symbol but when called a wolfram object with the
given rules.
```

EXAMPLE:

```
width: 12, initial-state: 0
      .
.....  .....
.    . .
 ...   ....
 .  ... .
   .  .    ...
.    ... .
 ... .     ..
 .    ... .
  ... .      .
. .    ...
    ... .  ..
.. .      . .
.   ...    .
 .. .  ...
  .    . .  ..
    .. .    . .
 . . ...      .
      .  ...
... .  .  ..
.  .      . .
 .  ...     .
   . .  ...
.      .  .  ..
```

```
 ...     . .
 .  ...     .
  .  .  ...
.    . .  ..
then last four rows repeat

also very nice starting with 0 instead of 1 (no obvious repeats):
(generate w 100 0)
............
.
 ..........
 .
 ..........
. .
   .........
.. .
.  ........
 .. .
 .  .......
  .. .
.. .   ......
    .. .
.. .  .....
.  .. .
 .. .   ....
 .  .. .
 .. .   ...
.. .  .. .
    .. .   ..
.. .  .. .
.  .. .    .
 .. .  ..
.  .. . ..
  .. .    .
.. .  ... .
   .. .  .
.. .  .  ..
.  .. . ..
 .. .     .
.     ...
 .... .  ..
. .     . .
   ....    .
... .   ...
```

SYNOPSIS:

```
(defwolfram make-wolfram-r25
    '(((1 1 1) 0) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1)
      ((0 1 1) 1) ((0 1 0) 0) ((0 0 1) 0) ((0 0 0) 1)))
```

28.5.428   wolfram/make-wolfram-r30

[wolfram] [Functions]

DATE:

 June 6th 2017, Edinburgh.

DESCRIPTION:

 This is the default Wolfgram object: #b00011110 = 30

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

 - the width of a row. Default = 100.
 - the initial state of the object. Default = 0.

RETURN VALUE:

 the function name as a symbol but when called a wolfram object with the
 given rules.

EXAMPLE:

```
;; so width: 12, initial-state: 0
(let ((w (make-wolfram-r30 12 0)))
  (generate w 100)
  (print-matrix w :row-number nil :off "." :on 'X)
  (print-matrix w :row-number nil))
      .
     ...
    ..  .
   .. ....
  ..  .  .
 .. .... ...
..  .    .
```

```
. .... ...
. .    ...  .
. .. ..  ...
. ..  . ..
. .... . .
. .     .....
. ..  ..
. . ... .
. .. .   ..
. . . .. .
. . . . ..
. . .......
. . .       .
. . ..    ..
. . . .  ..
. . . .... .
. . . .    .
. . .. . ..
. . . ...
. . . . . .
. . . . ...
. . . . .
. . . . ..
. . . . .
. . . . . ..
then last two rows repeat
```

SYNOPSIS:

```
(defwolfram make-wolfram-r30
    '(((1 1 1) 0) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.429   wolfram/make-wolfram-r62

[wolfram] [Functions]

DATE:

 June 7th 2017, Edinburgh.

DESCRIPTION:

 #b00111110 = 62

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

 - the width of a row. Default = 100.
 - the initial state of the object. Default = 0.

RETURN VALUE:

 the function name as a symbol but when called a wolfram object with the
 given rules.

EXAMPLE:

```
      .
     ...
    ..  .
   .. ....
  .. ..   .
 .. .. . ...
.. .. ....
. .. ..   .
... .. . ...
. .. ....
.... ..   .
.  .. . ...
.. .. ....
. .. ..   .
... .. . ...
.  .. ....
then the last 6 lines repeat
|#
(defwolfram make-wolfram-r62
    '(((1 1 1) 0) ((1 1 0) 0) ((1 0 1) 1) ((1 0 0) 1) ((0 1 1) 1)
      ((0 1 0) 1) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.430   wolfram/make-wolfram-r73

[wolfram] [Functions]

DATE:

 June 8th 2017, Edinburgh.

DESCRIPTION:

```
#b01001001 = 73
Suggested by Orestis Papadopoulos
```

ARGUMENTS:

```
none required
```

OPTIONAL ARGUMENTS:

```
- the width of a row. Default = 100.
- the initial state of the object. Default = 0.
```

RETURN VALUE:

```
the function name as a symbol but when called a wolfram object with the
given rules.
```

EXAMPLE:

```
       .
.....   ....
.   . . .  .
  .
.   ........
   . .     .
.    ....
   ... . ..
. . .
      ......
..... .    .
.   .  ..
  .   . .. .
then the last 2 lines repeat
|#
(defwolfram make-wolfram-r73
   '(((1 1 1) 0) ((1 1 0) 1) ((1 0 1) 0) ((1 0 0) 0) ((0 1 1) 1)
     ((0 1 0) 0) ((0 0 1) 0) ((0 0 0) 1)))
```

28.5.431   wolfram/make-wolfram-r82

[wolfram] [Functions]

DATE:

```
June 6th 2017, Edinburgh.
```

DESCRIPTION:

```
#b01010010 = 82
```

ARGUMENTS:

```
none required
```

OPTIONAL ARGUMENTS:

```
- the width of a row. Default = 100.
- the initial state of the object. Default = 0.
```

RETURN VALUE:

```
the function name as a symbol but when called a wolfram object with the
given rules.
```

EXAMPLE:

```
width: 12, initial-state: 0
non-repeating  but lovely patterns
      .
     . .
    .   .
    . . . .
   .       .
  . .     . .
 .   .   .
 . . . . .
 .         .
 .       . .
. .     .
  .   . .
  . . .   .
 .     . . .
. .   .
   . . .
  .       .
 . .     . .
 .   . .   .
  . .   . . .
 .   . .
  . .   .
 .     . . .
```

```
  .  .      .
.   .   . .
  . . . .   .
.       . .
  .      .   .
. .   . . .
    . .      .
    .   .   .
  . . . . . .
.
  .
. .
```

SYNOPSIS:

```
(defwolfram make-wolfram-r82
    '(((1 1 1) 0) ((1 1 0) 1) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 0)
      ((0 1 0) 0) ((0 0 1) 1) ((0 0 0) 0)))
```

28.5.432   wolfram/make-wolfram-r93

[wolfram] [Functions]

DATE:

 June 7th 2017, Edinburgh.

DESCRIPTION:

 #b01011101 = 93

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

 - the width of a row. Default = 100.
 - the initial state of the object. Default = 0.

RETURN VALUE:

 the function name as a symbol but when called a wolfram object with the
 given rules.

EXAMPLE:

```
        .
..... ......
.    ...     .
... . .... .
. ..... ...
...    .. . .
. ... .....
... ...     .
. ... .... .
... ... ...
. ... .. . .
... ........
. ...       .
... ...... .
. ...    ...
... .... . .
. ... .....
... .. .   .
. ........ .
...      ...
. ...... . .
...    .....
. .... .   .
... ..... .
. ... .   ...
........ . .
.      .....
..... .   .
.    ..... .
.... .   ...
. ..... . .
.. .  .....
...... .   .
.     ..... .
then the last 5 lines repeat
|#
(defwolfram make-wolfram-r93
   '(((1 1 1) 0) ((1 1 0) 0) ((1 0 1) 0) ((1 0 0) 1) ((0 1 1) 1)
     ((0 1 0) 1) ((0 0 1) 0) ((0 0 0) 1)))
```

28.5.433   wolfram/print-matrix

[wolfram] [Methods]

DATE:

 June 6th 2017, Edinburgh

DESCRIPTION:

 Print the state of the cells to the terminal. On and off could
 theoretically be any object but of course single characters are
 clearest. See the defwolfram definitions below for examples.

ARGUMENTS:

 - the wolfram object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :stream. The stream to print to. Default t which means it prints to the
   terminal.
 - :on. what to print when a cell has the value 1. Default: the . character
 - :off. what to print when a cell has the value 0. Default: space
 - :row-number: whether to print row numbers. Default t.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod print-matrix ((w wolfram) &key (stream t) (on #\.) (off #\ )
                                          (row-number nil))
```


28.5.434   wolfram/try-all

[wolfram] [Functions]

DATE:

 June 10th 2017, Edinburgh

DESCRIPTION:

 Try all of the Wolfram rules defined via defwolfram. An instance of each
 rule set defined will be created, the rows will be generated, and
 print-matrix will be called. This allows you to see and compare the results
 of each rule set using the same parameters.

ARGUMENTS:

 none required

OPTIONAL ARGUMENTS:

 keyword arguments:
 :width. The width slot (number of columns) for the Wolfram objects. Any
 positive ingeger. Default = 100.
 :initial-state. The initial-state (i.e. default) slot for the Wolfram
 objects. 0 or 1. Default = 0.
 :start. The value of the middle cell of the first row. See generate method
 for more details. 0 or 1. Default = 1.
 :generations. The number of rows to generate. Any positive integer. Default
 = 30.
 :stream, :on, :off, :row-number: see print-matrix method.

RETURN VALUE:

 T

SYNOPSIS:

```
(defun try-all (&key (width 100) (initial-state 0) (start 1) (generations 30)
                (stream t) (on #\.) (off #\ ) (row-number nil))
```

28.5.435   circular-sclist/at-start

[circular-sclist] [Methods]

DESCRIPTION:

 Determines whether the pointer for the given circular-sclist object is at
 the head of its list.

ARGUMENTS:

 - A circular-sclist object.

RETURN VALUE: EXAMPLE:

```
;; At creation the pointer is located at the start of the list
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (at-start cscl))
```

```
=> T
```

```
;; Retrieve a number of the items using get-next, then determine whether the
;; pointer is located at the start of the list
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 7 do (get-next cscl))
  (at-start cscl))
=> NIL
```

SYNOPSIS:

```
(defmethod at-start ((cscl circular-sclist))
```

28.5.436   circular-sclist/cycle-repeats

[circular-sclist] [Classes]

NAME:

```
 rthm-chain

 File:              cycle-repeats.lsp

 Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                    circular-sclist -> cycle-repeats

 Version:           1.0.11

 Project:           slippery chicken (algorithmic composition)

 Purpose:           class used in rthm-chain

 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:     4th February 2010

 $$ Last modified: 12:16:23 Mon May 23 2016 BST

 SVN ID: $Id$
```

28.5.437   circular-sclist/get-current

[circular-sclist] [Methods]

DESCRIPTION:

Return the current list element without incrementing the current slot.

ARGUMENTS:

 - The circular-sclist object

RETURN VALUE:

 The current list element, which can be of any type.

EXAMPLE:

```
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop for i below 10 do
     (print (if (oddp i) (get-next cscl) (get-current cscl)))))
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop for i below 10
     collect (if (oddp i) (get-next cscl) (get-current cscl))))
=>
(0 0 1 1 2 2 3 3 4 4)
```

SYNOPSIS:

```
(defmethod get-current ((cscl circular-sclist))
```

28.5.438   circular-sclist/get-last

[circular-sclist] [Methods]

DESCRIPTION:

 Return the the most recent item retrieved in a circular-sclist object.

ARGUMENTS:

 - A circular-sclist object.

RETURN VALUE:

 An item from the given circular-sclist object.

EXAMPLE:

```
;; Retrieves the final item in the list at creation
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (get-last cscl))
=> 4

;; Get and print a number of items from the list using get-next, then return
;; the most recent item retrieved using get-last
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 7 do (print (get-next cscl)))
  (get-last cscl))
=> 1
```

SYNOPSIS:

```
(defmethod get-last ((cscl circular-sclist))
```

28.5.439   circular-sclist/get-next

[circular-sclist] [Methods]

DESCRIPTION:

Get the next item in a given circular-sclist object. The class
automatically keeps track of the last item retrieved. If the final item of
the given circular-sclist object was the last item retrieved, the method
begins again at the beginning of the list.

ARGUMENTS:

- A circular-sclist object.

RETURN VALUE:

An item from the given circular-sclist object.

EXAMPLE:

```
;; Repeatedly calling get-next retrieves each subsequent item from the
;; given circular-sclist object. When the list has been exhausted, retrieval
;; begins again from the head of the list.
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 10 collect (get-next cscl)))
=>
(0 1 2 3 4 0 1 2 3 4)
```

SYNOPSIS:

```
(defmethod get-next ((cscl circular-sclist))
```

28.5.440   circular-sclist/make-cscl

[circular-sclist] [Functions]

DESCRIPTION:

 Create a circular-sclist object from a specified list of items. The items
 themselves may also be lists.

ARGUMENTS:

 - A list.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :id. A symbol that will be used as the ID for the created circular-sclist
   object. Default = NIL.
 - :bounds-alert. T or NIL to indicate whether or not to print a warning if
   when an attempt is made to access the object using an out-of-bounds index
   number (i.e., not enough elements in the list). T = print a
   warning. Default = T.
 - :copy. T or NIL to indicate whether the given data list should be copied
   (any slippery-chicken class instances will be cloned), with subsequent
   modifications being applied to the copy. T = copy. Default = T.

RETURN VALUE:

 A circular-sclist object.

EXAMPLE:

```
;; Returns a circular-sclist object with ID of NIL, bounds-alert=T and copy=T
;; by default
(make-cscl '(1 2 3 4))

=>
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 4, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
```

```
data: (1 2 3 4)

;; Can be created using nested lists
(let ((cscl (make-cscl '((1 (4 5 6))
                         (2 (7 8 9))
                         (3 (10 11 12))))))
  (data cscl))

=> ((1 (4 5 6)) (2 (7 8 9)) (3 (10 11 12)))

;; Setting the ID
(make-cscl '(1 2 3 4) :id 'test-cscl)

=>
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 4, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: TEST-CSCL, tag: NIL,
data: (1 2 3 4)

;; By default, attempts to access the object with an out-of-bounds index result
;; in a warning being printed
(let ((cscl (make-cscl '(1 2 3 4))))
  (get-nth 11 cscl))

=>
NIL
WARNING: sclist::sclist-check-bounds: Illegal list reference: 11
(length = 4) (sclist id = NIL)

;; This can be suppressed by creating the object with :bounds-alert set to NIL
(let ((cscl (make-cscl '(1 2 3 4) :bounds-alert nil)))
  (get-nth 11 cscl))

=> NIL
```

SYNOPSIS:

```
(defun make-cscl (list &key (id nil) (bounds-alert t) (copy t))
```

28.5.441   circular-sclist/popcorn

[circular-sclist] [Classes]

NAME:

```
assoc-list

File:             popcorn.lsp

Class Hierarchy:  named-object -> linked-named-object -> sclist ->
                  circular-sclist -> popcorn

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Inspired by popping popcorn, generate a series of values
                  ranging between > 0.0 and <= 1.0 by (optionally fixed)
                  random selection.  Given 1 or more starting values (not
                  zero) we generate tendentially increasing new values
                  until we reach 1.0.  This is not a linear process,
                  rather, we get spike values that increase the average
                  value and thus increase the chance of further spikes.

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    3rd February 2011 (Ko Lanta, Thailand)

$$ Last modified:  12:06:22 Sat Dec 14 2019 CET

SVN ID: $Id$
```

28.5.442   popcorn/fit-to-length

[popcorn] [Methods]

DESCRIPTION:

```
Change the length of the list of kernels contained in a given popcorn
object by adding or removing items at regular intervals. If adding items,
linear interpolation will be used.

NB: The new length must be between 1 and 1 less than double the original
    length.
```

ARGUMENTS:

```
- A popcorn object.
- An integer that is the new length of the list of the KERNELS slot of the
  given popcorn object.
```

RETURN VALUE:

 Returns the integer that is the new length of the KERNELS slot.

EXAMPLE:

```
(let ((ppcn (make-popcorn '(0.01 0.02) :min-spike 3.0 :max-spike 5.0)))
  (fit-to-length ppcn 100))
```

```
=> 100
```

SYNOPSIS:

```
(defmethod fit-to-length ((pc popcorn) length)
```

28.5.443   popcorn/get-kernel

[popcorn] [Methods]

DESCRIPTION:

 Generate the next value for the KERNELS slot of a given popcorn object and
 change the internal state, with the help of the get-kernel-aux method.

 This method is called automatically from within the heat method.

ARGUMENTS:

 - A popcorn object.

RETURN VALUE:

 The next value for the given popcorn object's KERNEL slot.
 Returns NIL when the kernel value is > 1.0.

SYNOPSIS:

```
(defmethod get-kernel ((pc popcorn))
```

28.5.444   popcorn/heat

[popcorn] [Methods]

DESCRIPTION:

Generate a series of values for the KERNELS slot of a popcorn object,
ranging between >0.0 and <= 1.0, by (optionally fixed) random selection.
If calling heat explicitly on a previously heated object, all kernels and
associated data will be deleted before being regenerated.

Taking the one or more starting values of the popcorn object, the method
generates tendentially increasing new values until it reaches 1.0. This is
not a linear process; rather, the method produces spike values based on the
min-spike and max-spike values of the given popcorn object that increase
the average value and thus increase the chance of further spikes.

NB: This method is called within the initialize-instance for the popcorn
    object, and as such is not necessarily needed to be accessed directly
    by the user.

ARGUMENTS:

 - An popcorn object.

RETURN VALUE:

 Returns the popcorn object with a newly generated list of 'kernel' values.

EXAMPLE:

```
(let ((ppcn (make-popcorn '(0.01 0.02) :min-spike 3.0 :max-spike 5.0)))
  (print ppcn)
  (setf (min-spike ppcn) 4.0)
  (heat ppcn))
=>
POPCORN: kernels: (0.01 0.02 0.016648924 0.018915312 0.016573396
                   0.017766343 0.018711153 0.017729789 0.017080924
                   0.018266398 0.018132625 0.019022772 0.017662765
[...]
POPCORN: kernels: (0.01 0.02 0.015828498 0.015408514 0.015781755 0.01670348
                   0.019892192 0.017849509 0.016623463 0.019682804 0.017869182
                   0.019521425 0.017451862 0.017689057 0.01758664 0.01863435
[...]
```

SYNOPSIS:

```
(defmethod heat ((pc popcorn))
```


28.5.445   popcorn/make-popcorn

[popcorn] [Functions]

DESCRIPTION:

```
Make a popcorn object. This method uses the heat method internally to
generate a series of decimal values ('kernels'), ranging between >0.0 and
<= 1.0, by (optionally fixed) random selection.

Taking the one or more starting values, the method generates tendentially
increasing new values until it reaches 1.0. This is not a linear process;
rather, the method produces spike values based on the min-spike and
max-spike values specified, which increase the average value of the kernels
generated so far and thus increase the chance of further spikes.
```

ARGUMENTS:

```
- A list of at least two decimal numbers from which the 'kernel' values
  will be generated. These values must be >0.0 and <1.0.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :id. An optional ID for the popcorn object to be created. Default = NIL.
- :fixed-random. T or NIL to indicate whether the 'kernel' values generated
  by the subsequent heat method are to be based on a fixed random seed.
  T = fixed random. Default = T.
- :max-spike. A decimal number that is the highest possible 'spike' value
  that the heat method may produce when generating the 'kernel'
  values. This is a sudden high value that will itself not be present in
  the final data, but will go towards skewing the mean, thus increasing the
  kernel values more rapidly and increasing the chance of more spikes
  occurring. Default = 4.0.
- :min-spike. A decimal number that is the lowest possible 'spike' value
  that the heat method may produce when generating the 'kernel'
  values. This is a sudden high value that will itself not be present in
  the final data, but will go towards skewing the mean, thus increasing the
  kernel values more rapidly and increasing the chance of more spikes
  occurring. Default = 2.0.
```

RETURN VALUE:

```
- A popcorn object.
```

EXAMPLE:

```
(make-popcorn '(0.02 0.03) :max-spike 4.2 :min-spike 3.7)
```

```
=>
POPCORN: kernels: (0.02 0.03 0.025828497 0.02540851 0.02578175 0.026703479
                   0.029892191 0.027849507 0.026623461 0.029682804 0.02786918
                   0.029521424 0.02745186 0.027689056 0.02758664 0.028634349
                   0.028176062 0.028434621 0.028410202 0.02834666 0.027676953
                   0.027972711 0.027877634 0.028453272 0.027664827 0.029336458
                   0.028315568 0.029327389 0.10877271 0.032779325 0.095442966
                   0.10383448 0.03631042 0.054371007 0.0775562 0.057371408
                   0.05496178 0.10499479 0.048501145 0.09311144 0.07531821
                   0.08538791 0.05866453 0.06692247 0.052130517 0.09605096
                   0.102914646 0.061326876 0.09510137 0.0927515 0.08405721
                   0.09921508 0.1054862 0.09474778 0.07701611 0.069283865
                   0.082345024 0.090727165 0.081423506 0.0918279 0.06942183
                   0.09431985 0.0790893 0.07795428 0.061114937 0.21615848
                   0.17666964 0.09314137 0.11025161 0.1909036 0.23906681
                   0.17467138 0.22562174 0.1757016 0.16630511 0.23570478
                   0.18461326 0.2358803 0.14396386 0.121555254 0.082086496
                   0.094552115 0.08456006 0.10379071 0.113467366 0.12590313
                   0.2211197 0.2096048 0.19645368 0.17204309 0.18469864
                   0.14422922 0.20209482 0.11207011 0.1176545 0.22522071
                   0.23593009 0.13767788 0.1589861 0.23501754 0.14337942
                   0.14403008 0.3852736 0.19077776 0.15493082 0.15311162
                   0.31107113 0.10612649 0.36018372 0.31991273 0.17881061
                   0.2653634 0.26506728 0.31478146 0.31331018 0.33569553
                   0.3001081 0.1574295 0.4698523 0.12513468 0.2010088
                   0.17438973 0.24960503 0.27139995 0.31985858 0.14607468
                   0.34586 0.52092844 0.5461051 0.33965456 0.24476483
                   0.45786726 0.23932996 0.18096672 0.5287333 0.45701692
                   0.58791053 0.5219719 0.39459002 0.56624746 0.37368405
                   0.21688993 0.3374743 0.6648663 0.44353223 0.16596928
                   0.3590309 0.17943183 0.673855 0.6455428 0.21892962
                   0.31195784 0.37920266 0.73120433 0.713979 0.5987564
                   0.29621923 0.5414667 0.64287895 0.56254905 0.514681
                   0.3153673 0.52838445 0.71745664 0.8074915 0.47637874
                   0.409207 0.49155992 0.777411 0.6339724 0.3673042 0.5411029
                   0.6993387 0.3566729 0.49429625 0.89963627 0.36773333
                   0.575006 0.74177176 0.53539884 0.4392826 0.45671058
                   0.2824728 0.60876155 0.2798523 0.47930354)
total: 44.67911, numk: 186, mink: 0.02, maxk: 0.89963627
min-spike: 3.7, max-spike: 4.2, fixed-random: T, mean: 0.24021028
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (0.02 0.03)
```

SYNOPSIS:

```
(defun make-popcorn (starting-values &key (id nil) (fixed-random t)
                     (max-spike 4.0) (min-spike 2.0))
```

28.5.446   popcorn/scale

[popcorn] [Methods]

DESCRIPTION:

Scale the list of number values in the KERNEL slot of a given popcorn
object to a new range using specified maximum value and optional minimum
value.

NB: This method does not change the internal state of the given popcorn
    object except for the KERNELS slot.

ARGUMENTS:

 - A popcorn object.
 - A number that is the new maximum value for the scaled list.

OPTIONAL ARGUMENTS:

 - A number that is the new minimum value for the scaled list.

RETURN VALUE:

 The new contents of the given popcorn object's KERNELS slot after scaling.

EXAMPLE:

```
;; Specifying a new maximum value only
(let ((ppcn (make-popcorn '(0.01 0.02) :min-spike 3.0 :max-spike 5.0)))
  (scale ppcn 10.0))

=> (0.0 0.10578585 0.061657257 0.057214428 0.061162785 0.070913345 0.10464539
    0.08303669 0.07006687 0.102430366 0.08324481 0.1007232 0.07883015
    0.08133934 0.0802559 0.09133919 0.08649117 0.08922634 0.08896804 0.08829583
    0.081211284 0.084339984 0.08333421 0.089423634 0.08108301 0.09876652
    0.08796693 0.09867058 0.6236897 0.1155461 0.5345579 0.59066933 0.13915743
    0.2599228 0.4149548 0.27998555 0.2638731 0.59842795 0.22067288 0.51896775
    0.39999005 0.467323 0.28863218 0.3438505 0.24494135 0.5386234 0.58451873
    0.30643454 0.53227377 0.516561 0.458425 0.55978096 0.60171384 0.52990943
```

```
    0.41134343 0.35964036 0.44697618 0.5030249 0.4408143 0.51038516 0.36056283
    0.527048 0.42520615 0.41761667 0.30501738 1.2049319 0.9747751 0.48793882
    0.5876642 1.0577364 0.3861802 0.8744062 1.1359217 0.87969404 0.8314643
    1.1876756 0.92543525 1.1885763 0.7167921 0.60177433 0.39919102 0.46317393
    0.4118872 0.5105933 0.56026125 0.62409085 1.112814 1.0537107 0.98620933
    0.86091584 0.92587364 0.71815413 1.0151639 0.5530894 0.5817527 1.1338633
    1.1888319 0.68452775 0.79389757 1.1841481 0.7137923 0.717132 2.3223429
    1.0770427 0.8475252 0.83587736 1.847246 0.53504527 2.1616995 1.9038564
    1.0004206 1.5545927 1.552697 1.8710022 1.861582 2.004909 1.7770531
    0.86352354 1.5642304 0.63962364 1.0099988 0.8800593 1.2472185 1.353609
    1.5901572 0.7418409 1.7170814 3.3055406 3.700319 2.1952631 1.5035022
    3.057052 1.4638814 1.0384043 1.1837897 3.0469408 2.803617 3.2614107
    2.4135168 3.556123 2.27436 1.2306889 2.0333362 4.5721135 2.941492 0.8966192
    2.3189502 0.9958008 4.344239 4.3671365 1.2785082 1.952021 2.438866
    4.3924103 4.46023 3.7118216 1.746746 3.3397071 3.9984121 3.4766433 3.165725
    1.8711188 3.2547336 4.482818 3.3188024 2.6877654 2.3054938 2.7741609
    3.796511 3.5835814 2.064381 3.0545063 3.9559705 2.0038147 2.7878509
    3.4762452 1.9495757 2.9715302 3.7937658 2.7762475 2.3023481 2.3882763
    1.5292001 3.1379611 1.5162798 2.4996707 4.362247 3.1643825 2.902113
    1.5552855 3.569274 3.6554635 3.8193665 3.2386634 5.418084 1.488541
    4.5816646 4.1958213 2.411787 2.6187074 3.1729605 2.959683 2.3334894
    5.325289 3.2408857 4.67207 3.0460484 6.0358443 6.879726 3.3280933 5.5901675
    1.8741251 3.5842674 4.855096 6.005389 1.7205821 3.8116035 3.439082 5.024595
    2.205073 4.140361 1.8645307 2.511795 5.744685 2.0451677 2.311025 6.787981
    6.533982 3.840785 2.2128632 6.444055 2.7525501 8.19589 7.3742037 2.5753407
    8.9812355 3.0030684 5.501138 6.7223954 4.8878922 3.2250557 2.3134975
    8.762646 3.072827 7.0158014 7.426256 5.388799 10.0 7.367759 7.078608
    8.373905 9.210589 7.072851 2.7709346 7.233898)
```

```
;; Using both a new maximum and new minimum value
(let ((ppcn (make-popcorn '(0.01 0.02) :min-spike 3.0 :max-spike 5.0)))
  (scale ppcn 8.0 5.0))
```

```
=> (5.0 5.031736 5.018497 5.017164 5.0183487 5.021274 5.0313935 5.024911
    5.02102 5.0307293 5.0249734 5.030217 5.023649 5.0244017 5.024077 5.027402
    5.0259476 5.0267677 5.0266905 5.026489 5.0243635 5.025302 5.025 5.026827
    5.024325 5.02963 5.02639 5.029601 5.187107 5.0346637 5.1603675 5.177201
    5.041747 5.0779767 5.1244864 5.083996 5.079162 5.179528 5.0662017 5.15569
    5.119997 5.140197 5.08659 5.103155 5.0734825 5.161587 5.1753554 5.0919304
    5.1596823 5.1549683 5.1375275 5.1679344 5.1805143 5.1589727 5.123403
    5.107892 5.134093 5.1509075 5.132244 5.1531157 5.108169 5.1581144 5.127562
    5.125285 5.091505 5.3614798 5.292433 5.146382 5.176299 5.317321 5.1158543
    5.262322 5.3407764 5.2639084 5.2494392 5.3563027 5.277631 5.356573 5.215038
    5.1805325 5.119757 5.1389523 5.123566 5.153178 5.1680784 5.1872272 5.333844
    5.3161135 5.2958627 5.2582746 5.277762 5.2154465 5.304549 5.165927
```

```
5.1745257 5.340159 5.3566494 5.2053585 5.238169 5.3552446 5.2141376
5.2151394 5.696703 5.323113 5.2542577 5.2507634 5.554174 5.1605134 5.64851
5.571157 5.300126 5.4663777 5.465809 5.5613008 5.5584745 5.601473 5.533116
5.259057 5.4692693 5.191887 5.3029995 5.2640176 5.3741655 5.4060826
5.477047 5.2225523 5.5151243 5.991662 6.110096 5.658579 5.4510508 5.9171157
5.4391646 5.3115215 5.355137 5.9140825 5.8410854 5.978423 5.7240553
6.066837 5.682308 5.369207 5.610001 6.3716345 5.8824477 5.2689857 5.6956854
5.2987404 6.303272 6.310141 5.3835526 5.5856066 5.73166 6.3177233 6.338069
6.1135464 5.524024 6.001912 6.199524 6.042993 5.9497175 5.5613356 5.9764204
6.3448453 5.9956408 5.8063297 5.691648 5.832248 6.138953 6.075074 5.619314
5.916352 6.1867914 5.6011443 5.836355 6.0428734 5.5848727 5.891459
6.1381297 5.8328743 5.6907043 5.716483 5.4587603 5.9413886 5.454884
5.7499013 6.3086743 5.9493146 5.870634 5.4665856 6.070782 6.096639 6.14581
5.971599 6.6254253 5.4465623 6.3744993 6.258746 5.723536 5.785612 5.951888
5.887905 5.700047 6.5975866 5.9722657 6.401621 5.9138145 6.8107533 7.063918
5.998428 6.6770506 5.5622377 6.07528 6.4565287 6.8016167 5.516175 6.1434813
6.0317245 6.5073786 5.661522 6.2421083 5.559359 5.7535386 6.723406 5.61355
5.6933074 7.0363946 6.9601946 6.1522355 5.663859 6.9332166 5.825765
7.4587674 7.212261 5.772602 7.694371 5.9009204 6.6503415 7.016719 6.4663677
5.967517 5.6940494 7.6287937 5.9218483 7.1047406 7.227877 6.61664 8.0
7.210328 7.123583 7.5121717 7.763177 7.1218557 5.8312807 7.17017)
```

SYNOPSIS:

```
(defmethod scale ((pc popcorn) max &optional (min 0.0) ignore1 ignore2)
```

28.5.447   circular-sclist/recurring-event

[circular-sclist] [Classes]

NAME:

```
 rthm-chain

 File:            recurring-event.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  circular-sclist -> recurring-event

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         class used in rthm-chain
                  This class allows for the periodic/cyclic return of given
```

```
                    data.  It is intended for situations where you want to
                    do/collect something every several events, but the cycle
                    period changes.  E.g. (the data slot is) something like
                    '((2 3) (3 2) (5 3) (8 2)) which means every two events
                    three times, then every 3 events twice, every 5 events
                    thrice etc.

                    If you want to return specific data on these cycle
                    points, provide it in the return-data slot, with the
                    indices into this data in the return-data-cycle slot.

                    simple example, without return-data
                    (let* ((re (make-re '((2 3) (3 2) (5 3) (8 2))
                                        :return-data nil
                                        :return-data-cycle nil)))
                      (loop repeat 100 collect (on-it re)))

                    => (NIL NIL T NIL T NIL T NIL NIL T NIL NIL T NIL NIL NIL
                        NIL T NIL NIL NIL NIL T NIL NIL NIL NIL T NIL NIL NIL
                        NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL T NIL T
                        NIL T NIL T NIL NIL T NIL NIL T NIL NIL NIL NIL T NIL
                        NIL NIL NIL T NIL NIL NIL NIL T NIL NIL NIL NIL NIL
                        NIL NIL T NIL NIL NIL NIL NIL NIL NIL T NIL T NIL T
                        NIL T NIL NIL T NIL NIL T NIL)

                    (let* ((re (make-re '((2 3) (3 2) (5 3) (8 2))
                                        ;; the data about to be collected
                                        :return-data '(a b c d)
                                        ;; the indices into the data; this
                                        ;; means we'll return A (nth 0)
                                        ;; thrice, D (nth 3) twice, C once,
                                        ;; and B 5x
                                        :return-data-cycle
                                        '((0 3) (3 2) (2 1) (1 5)))))
                      (loop repeat 100 collect (get-it re)))

                    => (NIL NIL A NIL A NIL A NIL NIL D NIL NIL D NIL NIL NIL
                        NIL C NIL NIL NIL NIL B NIL NIL NIL NIL B NIL NIL NIL
                        NIL NIL NIL NIL B NIL NIL NIL NIL NIL NIL NIL B NIL B
                        NIL A NIL A NIL NIL A NIL NIL D NIL NIL NIL NIL D NIL
                        NIL NIL NIL C NIL NIL NIL NIL B NIL NIL NIL NIL NIL
                        NIL NIL B NIL NIL NIL NIL NIL NIL NIL B NIL B NIL B
                        NIL A NIL NIL A NIL NIL A NIL)

 Author:            Michael Edwards: m@michael-edwards.org
```

```
 Creation date:    4th February 2010

 $$ Last modified:  10:39:52 Sat Jun  6 2020 CEST

 SVN ID: $Id$
```

28.5.448   recurring-event/get-it

[recurring-event] [Methods]

DESCRIPTION:

```
 Get the next element from the return-data. This method is most effective
 when called repeatedly (e.g. within a loop) when the return-data and
 return-data-cycle slots have been set. In those cases the return-data-cycle
 element will be used as look-up into return-data. If no return-data has
 been specified, then the element itself will be returned.
```

ARGUMENTS:

```
 - A recurring-event object.
```

RETURN VALUE:

```
 Data from the return-data slot (or the return-data-cycle element) when
 we're on a boundary, otherwise NIL.
```

EXAMPLE:

```
;;; Used together with return-data
(let ((re (make-re '((2 3) (3 2) (5 3) (8 2))
                   :return-data '(a b c d)
                   :return-data-cycle '((0 3) (3 2) (2 1) (1 5)))))
  (loop repeat 50 collect (get-it re)))

=> (NIL NIL A NIL A NIL A NIL NIL D NIL NIL D NIL NIL NIL NIL C NIL NIL NIL NIL
    B NIL NIL NIL NIL B NIL NIL NIL NIL NIL NIL NIL B NIL NIL NIL NIL NIL NIL
    NIL B NIL B NIL A NIL A)

;;; Used without return-data
(let ((re (make-re '((2 3) (3 2) (5 3) (8 2))
                   :return-data-cycle '((0 3) (3 2) (2 1) (1 5)))))
  (loop repeat 50 collect (get-it re)))

=> (NIL NIL 0 NIL 0 NIL 0 NIL NIL 3 NIL NIL 3 NIL NIL NIL NIL 2 NIL NIL NIL NIL
```

```
    1 NIL NIL NIL NIL 1 NIL NIL NIL NIL NIL NIL NIL 1 NIL NIL NIL NIL NIL NIL
    NIL 1 NIL 1 NIL 0 NIL 0)
```

SYNOPSIS:

```
(defmethod get-it ((re recurring-event))
```

28.5.449   recurring-event/make-re

[recurring-event] [Functions]

DESCRIPTION:

```
 Make an instance of a recurring-event object, which allows for the
 periodic/cyclic return of given data. The recurring-event object is
 intended for situations in which the user would like to perform an action
 or collect data every several events, but with a varying cycle period.
```

ARGUMENTS:

```
 - A list of two-item lists that indicate the period pattern by which the
   action or data collection is to be performed. For example, a value such
   as '((2 3) (3 2) (5 3) (8 2)) will result in the action being performed
   every 2 events three times, then every 3 events twice, every 5 events
   thrice etc.
```

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :return-data. If the recurring-event object is to be used to collect
   data, that data can be specified in this slot, with the indices into this
   data in the return-data-cycle slot. The return-data and return-data-cycle
   slots must be used together.
 - :return-data-cycle. If data is specified using :return-data, the indices
   into that data must be specified here. For example, the value
   '((0 3) (3 2) (2 1) (1 5)) will the data item at (nth 0) thrice, that at
   (nth 3) twice, that at (nth 2) once, and that at (nth 1) five times.
 - :id. An optional ID can also be specified for the recurring-event object
   created.
```

RETURN VALUE:

```
 A recurring-event object.
```

EXAMPLE:

```
;;; Simple usage with no specified data
(make-re '((2 3) (3 2) (5 3) (8 2)))
=>
RECURRING-EVENT: current-period: 2, current-repeats: 3
                 pcount: -1, rcount: 0
                 return-data: NIL, return-data-cycle: NIL
CIRCULAR-SCLIST: current 1
SCLIST: sclist-length: 4, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: ((2 3) (3 2) (5 3) (8 2))

;;; Usage with specified :return-data and :return-data-cycle
(make-re '((2 3) (3 2) (5 3) (8 2))
         :return-data '(a b c d)
         :return-data-cycle '((0 3) (3 2) (2 1) (1 5)))

=>
RECURRING-EVENT: current-period: 2, current-repeats: 3
                 pcount: -1, rcount: 0
                 return-data: (A B C D), return-data-cycle:
CYCLE-REPEATS: folded: ((0 3) (3 2) (2 1) (1 5))
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 11, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (0 0 0 3 3 2 1 1 1 1 1)
**************

CIRCULAR-SCLIST: current 1
SCLIST: sclist-length: 4, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: ((2 3) (3 2) (5 3) (8 2))
```

SYNOPSIS:

```
(defun make-re (data &key return-data return-data-cycle id)
```

28.5.450   recurring-event/on-it

[recurring-event] [Methods]

DESCRIPTION:

 Test to determine whether the method is currently at a period boundary. The

object keeps track of its own internal state and position counter. This
method is most effective when called repeatedly in a loop.

ARGUMENTS:

- A recurring-event object.

RETURN VALUE:

T or NIL.

EXAMPLE:

```
;;; Straightforward usage
(let ((re (make-re '((2 3) (3 2) (5 3) (8 2))
                   :return-data '(a b c d)
                   :return-data-cycle '((0 3) (3 2) (2 1) (1 5)))))
  (loop repeat 50 collect (on-it re)))

=> (NIL NIL T NIL T NIL T NIL NIL T NIL NIL T NIL NIL NIL NIL T NIL NIL NIL NIL
    T NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL
    NIL T NIL T NIL T NIL T)
```

SYNOPSIS:

```
(defmethod on-it ((re recurring-event))
```

28.5.451   circular-sclist/reset

[circular-sclist] [Methods]

DESCRIPTION:

Reset the pointer of a given circular-sclist object. The pointer is reset
to 0 by default, but the desired index may be specified using the optional
argument.

NB: An immediately subsequent get-next call will retrieve the item at the
    index to which the pointer is reset. An immediately subsequent
    get-last call will retrieve the item at the index one-less
    than the value to which the pointer is set.

ARGUMENTS:

- A circular-sclist object.

OPTIONAL ARGUMENTS:

  - An index integer to which the pointer for the given circular-sclist
    object should be reset.

RETURN VALUE:

 Returns T.

EXAMPLE:

```
;; Resets to 0 by default. Here: Get a number of items using get-next, reset
;; the pointer, and apply get-next again.
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 8 do (print (get-next cscl)))
  (reset cscl)
  (get-next cscl))
=> 0

;; Reset to a specified index
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 8 do (print (get-next cscl)))
  (reset cscl 3)
  (get-next cscl))
=> 3

;; By default, get-last will then retrieve the item at index one less than the
;; reset value
(let ((cscl (make-cscl '(0 1 2 3 4))))
  (loop repeat 8 do (print (get-next cscl)))
  (reset cscl 3)
  (get-last cscl))
=> 2
```

SYNOPSIS:

```
(defmethod reset ((cscl circular-sclist) &optional where (warn t))
```

28.5.452   circular-sclist/rthm-chain-slow

[circular-sclist] [Classes]

NAME:

 rthm-chain

```
File:              rthm-chain-slow.lsp

Class Hierarchy:   named-object -> linked-named-object -> sclist ->
                   circular-sclist -> rthm-chain-slow

Version:           1.0.11

Project:           slippery chicken (algorithmic composition)

Purpose:           class used in rthm-chain

Author:            Michael Edwards: m@michael-edwards.org

Creation date:     4th February 2010

$$ Last modified: 11:20:15 Mon Dec 17 2012 ICT

SVN ID: $Id$
```

28.5.453   sclist/combine

[sclist] [Methods]

DESCRIPTION:

Combine the contents of two given sclist objects into one list. NB This
changes the data list of a clone of the first argument by appending a
copy of the data list of the second argument i.e. it creates a wholly new
sclist object which it then returns.

ARGUMENTS:

- A first sclist object.
- A second sclist object.

RETURN VALUE:

Returns an sclist object.

EXAMPLE:

```
;; Combine the contents of two sclist objects to make a new one
(let ((scl1 (make-sclist '(0 1 2 3 4)))
      (scl2 (make-sclist '(5 6 7 8 9))))
```

```
  (combine scl1 scl2))
```

```
=>
SCLIST: sclist-length: 10, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (0 1 2 3 4 5 6 7 8 9)
```

SYNOPSIS:

```
(defmethod combine ((scl1 sclist) (scl2 sclist))
```

28.5.454   sclist/get-last

[sclist] [Methods]

DESCRIPTION:

 Returns the last element in the data list of a given sclist.

ARGUMENTS:

 - An sclist.

RETURN VALUE:

 The last object in the data list.

EXAMPLE:

```
(let ((al (make-assoc-list 'test '((jim beam)
                                   (four roses)
                                   (wild turkey)))))
  (get-last al))
```

```
=>
NAMED-OBJECT: id: WILD, tag: NIL,
data TURKEY
```

SYNOPSIS:

```
(defmethod get-last ((scl sclist))
```

28.5.455   sclist/get-nth

[sclist] [Methods]

DESCRIPTION:

 Get the nth element (zero-based) of data in a given sclist object.

ARGUMENTS:

 - An index integer.
 - An sclist object.

RETURN VALUE:

 Returns the item at index n within the given sclist object.

 Returns NIL and prints a warning if the specified index is greater than the
 number of items in the given list (minus 1).

EXAMPLE:

```
;; Get the 3th item from the given sclist object
(let ((scl (make-sclist '(cat dog cow pig sheep))))
  (get-nth 3 scl))

=> PIG

;; Returns NIL and prints a warning when the specified index is beyond the
;; bounds of the given list
(let ((scl (make-sclist '(cat dog cow pig sheep))))
  (get-nth 31 scl))

=>
NIL
WARNING: sclist::sclist-check-bounds: Illegal list reference: 31
(length = 5) (sclist id = NIL)
```

SYNOPSIS:

```
(defmethod get-nth (index (i sclist)))
```

28.5.456   sclist/intervals-mapper

[sclist] [Classes]

NAME:

```
intervals-mapper
```

File:	intervals-mapper.lsp
Class Hierarchy:	named-object -> linked-named-object -> sclist -> intervals-mapper
Version:	1.0.11
Project:	slippery chicken (algorithmic composition)
Purpose:	Implementation of a scale object that can initialize its complete pitch list from the interval structure of a given list of notes. Given the scale, it's a cinch to generate note sequences based on note offset patterns e.g.

```
                    (let ((s (make-intervals-mapper 'c0 '(d e gs as d ef g a
                                        bf cs d ef gf)))
                          (pat '(-1 2 4  3 6 -2 -1 2 6  7 3 6 2)))
                      (loop for p in pat collect
                           (data (intervals-mapper-note s p 4))))
                    -> (A3 EF4 F4 E4 BF4 F3 A3 EF4 BF4 D5 E4 BF4 EF4)
```

Author:	Michael Edwards: m@michael-edwards.org
Creation date:	August 3rd 2010 Edinburgh
$$ Last modified:	12:57:04 Fri Sep 23 2016 BST

```
SVN ID: $Id$
```

28.5.457   intervals-mapper/get-pitch-symbols

[intervals-mapper] [Methods]

DESCRIPTION:

Get the pitches contained in a given intervals-mapper object returned as a list of note-name symbols.

ARGUMENTS:

- An intervals-mapper object.

RETURN VALUE:

A list of note-name pitch symbols.

EXAMPLE:

```
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (get-pitch-symbols im))
```

```
=> (C0 D0 FS0 AF0 C1 CS1 F1 G1 AF1 B1 C2 CS2 E2 FS2 BF2 C3 E3 F3 A3 B3 C4 EF4
    E4 F4 AF4 BF4 D5 E5 AF5 A5 CS6 EF6 E6 G6 AF6 A6 C7 D7 FS7 AF7 C8 CS8 F8 G8
    AF8 B8 C9 CS9 E9 FS9)
```

SYNOPSIS:

```
(defmethod get-pitch-symbols ((im intervals-mapper) &optional ignore)
```

28.5.458   intervals-mapper/get-scale

[intervals-mapper] [Methods]

DESCRIPTION:

Create a scale (sequence of pitches) beginning with the specified starting
note and extending up to MIDI note 127 by cycling through the interval
structure of the STEPS slot.

The scale will only repeat at octaves if the interval structure of the list
of pitches passed at initialisation creates that result.

NB: This method is usually only called automatically at initialisation.

ARGUMENTS:

- An intervals mapper object.
- A note-name pitch symbol (e.g. 'c0) that is the pitch on which to begin
  the scale, and which is to be stored in the TONIC slot.

RETURN VALUE:

A list of the pitch objects in the new scale. These are also stored in the
SCALE-PITCHES slot.

EXAMPLE:

```
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
```

```
  (pitch-list-to-symbols (get-scale im 'd4)))
```

=> (D4 E4 AF4 BF4 D5 EF5 G5 A5 BF5 CS6 D6 EF6 FS6 AF6 C7 D7 FS7 G7 B7 CS8 D8 F8
    FS8 G8 BF8 C9 E9 FS9)

SYNOPSIS:

```
(defmethod get-scale ((im intervals-mapper) start-note)
```

28.5.459   intervals-mapper/get-steps

[intervals-mapper] [Methods]

DESCRIPTION:

 Extract the interval structure of the list of note-name pitch symbols
 passed as the data to the instance of the intervals-mapper object upon
 initialization. The interval structure is returned as a list of semitone
 values.

ARGUMENTS:

 - An intervals-mapper object.

RETURN VALUE:

 A list of integers that are the numbers of semitones between each
 consecutive pitch in the original data list.

EXAMPLE:

```
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (get-steps im))
```

=> (2 4 2 4 1 4 2 1 3 1 1 3)

SYNOPSIS:

```
(defmethod get-steps ((im intervals-mapper))
```

28.5.460   intervals-mapper/intervals-mapper-degree

[intervals-mapper] [Methods]
DATE:

14-Aug-2010

DESCRIPTION:

Return the scale degree number of a specified pitch class in relation to a
specified octave within the given intervals-mapper object.

To determine the scale degree number, this method begins at the first pitch
>= C in the specified octave and passes consecutively through each
subsequent pitch in the interval-mapper object's full scale, counting each
step until it first encounters the pitch class of the specified pitch.

If there are no more instances of the specified pitch class, the method
returns NIL.

The method takes as its pitch class a pitch object, which includes an
octave indicator. For the purposes of this method, solely the pitch-class
name is extracted from the pitch object.

ARGUMENTS:

 - An intervals-mapper object.
 - An instance of a pitch object whose pitch class is being sought.
 - An integer that is the octave in relationship to which the scale degree
   is sought.

OPTIONAL ARGUMENTS:

 - T or NIL to indication whether to return the position of the found pitch
   object within the complete scale list of the given intervals-mapper
   object rather than the scale degree. T = return the position.
   Default = NIL.

RETURN VALUE:

Returns an integer that is either the scale degree of the specified pitch
class in relation to the specified octave, counting from 1, or the position
of the found pitch object within the interval-mapper object's complete
scale.

Returns NIL if no instances of the specified pitch class are found.

EXAMPLE:

;;; The desired pitch class BF is found within the specified octave

```
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-degree im (make-pitch 'bf4) 4))
```

=> 6

```
;;; The desired pitch class B is found outside of the specified octave
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-degree im (make-pitch 'b4) 5))
```

=> 20

```
;;; Return the position instead
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-degree im (make-pitch 'b4) 5 t))
```

=> 45

```
;;; The desired pitch class BF is not present in any of the octaves beginning
;;; including and above the specified octave

(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-degree im (make-pitch 'bf4) 5))
```

=> NIL

SYNOPSIS:

```
(defmethod intervals-mapper-degree ((im intervals-mapper) pitch octave
                                    &optional return-nth)
```

28.5.461   intervals-mapper/intervals-mapper-note

[intervals-mapper] [Methods]

DESCRIPTION:

 Get the pitch object that constitutes the specified scale degree of the
 specified octave within an intervals-mapper object; or, if keyword argument
 <nth> is set to T, get the position of this pitch object within the full
 range of the intervals-mapper object's complete scale.

 As there is no concept of a tonic that repeats at octaves, degree 1 in any
 given octave is simply the first note >= the C in that octave.

 If a note-name pitch symbol is given for the keyword argument <tonic>, the

given intervals-mapper object's TONIC slot will be changed, and its
scale-pitches will be recalculated accordingly before getting the pitch.

ARGUMENTS:

 - An intervals-mapper object.
 - An integer that is the scale degree (1-based) of the desired pitch within
   the specified octave, counting from the first note of the scale above or
   on the C in that octave. If this number is higher than the number of
   pitches in the span of an octave, a pitch from a higher octave will
   accordingly be returned. Similarly, a negative number can also be given
   here to indicate that the pitch is to be collected from that many degrees
   below the specified octave.
 - An integer that indicates the octave from which the pitch is to be
   returned (e.g. 4 for the octave starting on middle C, 5 for the octave
   starting on the C above that etc.)

OPTIONAL ARGUMENTS:

 keyword arguments
 - :tonic. NIL or a note-name pitch symbol that is the new starting note for
   above which the intervals-mapper scale is to be re-mapped before the
   pitch is returned. This will usually be the lowest pitch on which the
   scale should start. If NIL, no changes will be made to the object's
   existing scale pitches before returning the desired pitch. Default = NIL.
 - :nth. T or NIL to indicate whether instead of returning the pitch object
   itself, the method should return an integer that is the position of that
   pitch within the full span of the interval-mapper object's complete
   scale. T = return the position. Default = NIL.

RETURN VALUE:

 A pitch object by default, or its position in the scale list if the :nth
 argument is set to T.

EXAMPLE:

```
;;; Returns a pitch object by default
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-note im 3 4))

=>
PITCH: frequency: 329.628, midi-note: 64, midi-channel: 0
       pitch-bend: 0.0
       degree: 128, data-consistent: T, white-note: E4
```

```
        nearest-chromatic: E4
        src: 1.2599211, src-ref-pitch: C4, score-note: E4
        qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
        micro-tone: NIL,
        sharp: NIL, flat: NIL, natural: T,
        octave: 4, c5ths: 0, no-8ve: E, no-8ve-no-acc: E
        show-accidental: T, white-degree: 30,
        accidental: N,
        accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E4, tag: NIL,
data: E4

;;; Used with negative degree numbers
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (data (intervals-mapper-note im -3 2)))

=> F1

;;; Use with a new tonic and setting nth to T to return the position
(let ((im (make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))))
  (intervals-mapper-note im 11 5 :tonic 'd1 :nth t))

=> 29
```

SYNOPSIS:

```
(defmethod intervals-mapper-note ((im intervals-mapper) degree octave
                                  &key tonic nth)
```

28.5.462   intervals-mapper/make-intervals-mapper

[intervals-mapper] [Functions]

DESCRIPTION:

Returns an intervals-mapper object starting with the specified pitch
('tonic') and using the interval structure of the specified list of
pitches, creating a complete pitch list from the interval structure of list
of pitches.

NB The pitches specified aren't necessarily used in the resulting complete
   pitch list; rather, their interval structure is mapped above the
   specified starting pitch and repeated upwards until the method reaches
   MIDI note 127.

NB The scale will only repeat at octaves if that is the interval structure
    of the list of pitches it is passed.

ARGUMENTS:

 - A note-name pitch symbol that is the starting pitch, (e.g. 'c0).
 - A list of note-name pitch symbols that provides the interval structure
   for the resulting scale.

RETURN VALUE:

 An intervals-mapper object.

EXAMPLE:

```
;;; A scale without repeating octaves:
(make-intervals-mapper 'c0 '(d e gs as d ef g a bf cs d ef gf))

=>
INTERVALS-MAPPER: steps: (2 4 2 4 1 4 2 1 3 1 1 3),
scale-pitches (pitch objects): (C0 D0 FS0 AF0 C1 CS1 F1 G1 AF1 B1 C2 CS2 E2 FS2
                                BF2 C3 E3 F3 A3 B3 C4 EF4 E4 F4 AF4 BF4 D5 E5
                                AF5 A5 CS6 EF6 E6 G6 AF6 A6 C7 D7 FS7 AF7 C8
                                CS8 F8 G8 AF8 B8 C9 CS9 E9 FS9)
tonic: C0
SCLIST: sclist-length: -1, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (D E GS AS D EF G A BF CS D EF GF)
```

SYNOPSIS:

```
(defun make-intervals-mapper (tonic notes)
```

28.5.463   sclist/limits

[sclist] [Methods]
DATE:

 January 12th 2019, Heidhausen

DESCRIPTION:

Get the lowest and highest values of the list

ARGUMENTS:

- an sclist object

RETURN VALUE:

two values: the minimum and maximum values in the list

SYNOPSIS:

(defmethod limits ((scl sclist))

28.5.464   sclist/make-sclist

[sclist] [Functions]

DESCRIPTION:

Create an sclist object with the specified list.

ARGUMENTS:

- A list of numbers or symbols.

OPTIONAL ARGUMENTS:

keyword arguments:
- :id. A symbol that will be the ID of the given sclist object.
  Default = NIL.
- :bounds-alert. T or NIL to indicate whether a warning should be issued
  when a request is given to set or get an out-of-bounds element (i.e. not
  enough elements in list). T = print warning. Default = NIL.
- :copy. T or NIL to indicate whether the data in the list should be copied
  (any slippery-chicken class instances will be cloned), with subsequent
  modifications being applied to the copy. T = copy. Default = T.

RETURN VALUE:

Returns an sclist object.

EXAMPLE:

```
;; Create a simple object with just a list of numbers
(make-sclist '(1 2 3 4 5 6 7))

=>
SCLIST: sclist-length: 7, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (1 2 3 4 5 6 7)

;; Create the same object and assign an ID to it
(make-sclist '(1 2 3 4 5 6 7) :id 'number-list)

=>
SCLIST: sclist-length: 7, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NUMBER-LIST, tag: NIL,
data: (1 2 3 4 5 6 7)
```

SYNOPSIS:

```
(defun make-sclist (list &key (id nil) (bounds-alert t) (copy t))
```

28.5.465   sclist/max-items

[sclist] [Methods]

DATE:

 September 21st 2018, Heidhausen

DESCRIPTION:

 A destructive method to reduce the number of items in the list to a maximum
 of the 2nd argument. The remaining elements can be from either the beginning
 of the list, the middle, or the end. If the list has fewer then or the same
 number of elements as the 2nd argument, the list will remain unchanged.

ARGUMENTS:

 - the sclist object
 - the number of elements the list should be reduced to (integer, >= 0)

OPTIONAL ARGUMENTS:

 - a symbol ('start 'middle or 'end) to indicate which elements remain. See
   examples below. Default = 'start

RETURN VALUE:

 the (potentially) modified sclist object (NB this method is destructive!)

EXAMPLE:

```
(max-items (make-sclist '(1 2 3 4 5 6 7)) 3)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (1 2 3)
**************

(max-items (make-sclist '(1 2 3 4 5 6 7)) 3 'end)


SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (5 6 7)
**************

(max-items (make-sclist '(1 2 3 4 5 6 7)) 3 'middle)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (3 4 5)
**************
```

SYNOPSIS:

```
(defmethod max-items ((scl sclist) max &optional (from 'start))
```

28.5.466   sclist/new-limits

[sclist] [Methods]

DATE:

 January 12th 2019, Heidhausen

DESCRIPTION:

 Rescale the numbers in the list to be within a new range.

ARGUMENTS:

 - an sclist object

OPTIONAL ARGUMENTS:

 - the new minimum value. Default = 0.0
 - the new maximum value. Default = 100.0

RETURN VALUE:

 the new data list

EXAMPLE:

```
(new-limits (make-sclist '(-1.4 5 6 2.3 7.3 -2.1)))
-->
(7.4468083 75.53192 86.17022 46.80851 100.0 0.0)
```

SYNOPSIS:

```
(defmethod new-limits ((scl sclist) &optional (new-min 0.0) (new-max 100.0))
```

28.5.467   sclist/pitch-seq

[sclist] [Classes]

NAME:

 pitch-seq

 File: pitch-seq.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  pitch-seq

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

Purpose: Implementation of the pitch-seq class. This describes
                      the pitch curves for a given rhythmic sequence. These
                      are normally simple lists of notes indicating pitch
                      height (and later mapped onto sets); chords are indicated
                      by placing a number in parentheses.

Author: Michael Edwards: m@michael-edwards.org

Creation date: 19th February 2001

$$ Last modified: 14:39:26 Sat Jul 11 2020 CEST

SVN ID: Id

28.5.468   pitch-seq/get-notes

[pitch-seq] [Methods]

DESCRIPTION:

This gets notes from the sets, limiting the notes used to the range of the
instrument and any other ranges defined in the slippery-chicken class. If
either the instrument or set are missing it just gets the relative pitches
we use to display a pitch sequence.

limit-high and limit-low are pitch objects. They are extra range definers
that are given to the slippery-chicken object to control the pitch curve of
an instrument over the duration of the whole piece. They always refer to
sounding pitches.

The order of operations for selecting pitches are as follows:

1) Limit the set object to the instrument's range.

2) Remove the notes that have already been selected for other instruments.
   This is where the slippery-chicken slot :instrument-hierarchy plays an
   important role. This can be skipped if the <avoid-used-notes> argument
   is nil.

3) If there is a subset with the same ID as the ID slot for the
   player, use only those pitches common to that subset and those in
   step 2. If not, try again using the subset-id of the instrument.

4) If the ratio between the number of pitches now available and the number
   of different numbers in the pitch-seq is less than the slippery-chicken
   slot pitch-seq-index-scaler-min*, add notes from those used by other

instruments until there are enough; the lowest number in the pitch-seq
will now select the lowest pitch in the set that is in the instrument's
range.

If however there are enough pitches without adding pitches already used
by other instruments, then where in the available pitches the lowest
number of the pitch-seq will be placed depends on whether the
prefers-notes slot of the instrument has been set to be high or low. If
high, then the highest number in the pitch-seq will result in the
highest pitch in the available pitches that is in the instrument's
range. If low, then the lowest number in the pitch-seq will result in
the lowest pitch in the available pitches that is in the instrument's
range. If the user hasn't set this slot, then the range of the pitch-seq
will correspond to the middle of the available pitches.

There are two caveats here if the instrument's prefers-notes slot is
NIL: 1) If the lowest number in the pitch-seq is 5 or higher, this will
have the same effect as the prefers-notes slot being high. Similarly, if
the lowest number is 1, it will have the same effect as the
prefers-notes slot being low. These two numbers (5 and 1) are actually
global slippery chicken configuration data: (get-sc-config
pitch-seq-lowest-equals-prefers-high) and (get-sc-config
pitch-seq-lowest-equals-prefers-low) so can be set using the
set-sc-config function.

* The question as to how many pitches are enough pitches before adding
used notes is determined by the pitch-seq-index-scaler-min argument,
which is by default 0.5 (in the slippery-chicken slot that's usually
used and passed to this method). As the pitch-seq notes must be offset
and scaled before they can be used as indices, there's a minimum scaler
that's considered acceptable; anything below this would result in more
notes being added.

5) If at this point there are no available pitches, the function will by
   default trigger an error and exit (see however pitch-seq-no-pitches-error
   in globals.lsp). This could happen if the value of set-limits, both high
   and low, took the available pitches outside of the instrument's range,
   for instance.

6) The pitch-seq numbers are now offset and scaled, then rounded in order
   to use them as indices into the pitch list. If a number is in
   parentheses then this is where the instrument's chord function would be
   called. As notes are selected, the set marks them as used for the next
   time around. Also, there's an attempt to avoid melodic octaves on
   adjacent notes; however, if the set is full of octaves this won't be
   possible; in that case a warning will be issued and the octave will be

used.

ARGUMENTS:

 - A pitch-seq object.
 - An instrument object.
 - A player object.
 - An sc-set object.
 - A hint pitch (ignored for now).
 - A pitch-object defining the highest possible note.
 - A pitch-object defining the lowest possible note.
 - The sequence number (for diagnostics).
 - The last note of the previous sequence, as a pitch object.
 - The lowest scaler that will be accepted before adding notes from those
   used; i.e., if the pitch-seq needs 6 notes and only 3 are available,
   there would be note repetition, but as this would create a scaler of 0.5,
   that would be acceptable
 - Whether to avoid lines jumping an octave in either direction (passed by
   the slippery chicken slot).
 - Whether to remove notes already chosen for other instruments before
   selecting notes for this one.

RETURN VALUE:

 Returns the list of pitch objects that forms the notes slot of the given
 pitch-seq

SYNOPSIS:

```
(defmethod get-notes ((ps pitch-seq) instrument player set hint-pitch limit-high
                       limit-low seq-num last-note-previous-seq
                       pitch-seq-index-scaler-min avoid-melodic-octaves
                       avoid-used-notes)
```

28.5.469   pitch-seq/invert

[pitch-seq] [Methods]

DESCRIPTION:

 Invert the pitch sequence contour attached to a given pitch-seq object. The
 inversion uses only the same numbers from the original pitch contour list.

ARGUMENTS:

- A pitch-seq object.

RETURN VALUE:

 A pitch-seq object.

EXAMPLE:

```
(let ((ps (make-pitch-seq '(pseq1 (1 2 1 3 4 7)))))
  (data (invert ps)))

=> (7 4 7 3 2 1)
```

SYNOPSIS:

```
(defmethod invert ((ps pitch-seq) &optional ignore1 ignore2)
```

28.5.470   pitch-seq/make-pitch-seq

[pitch-seq] [Functions]

DESCRIPTION:

 Create a pitch-seq object.

 This function can be either called with one argument, consisting of a
 two-item list, in which the first item is the pitch-seq ID and the second
 is a list of numbers representing the pitch curve of the intended pitch
 sequence; or it can be created with two arguments, the first of which being
 the list of numbers representing the pitch curve and the second being the
 pitch-seq's ID.

 NB We can assign a pitch-seq exclusively to particular instruments in the
 ensemble simply by passing their names as symbols along with the curve
 data. See below for an example.

ARGUMENTS:

- A two-item list, of which the first item is a symbol to be used as the
  object's ID, and the second is a list of integers representing the general
  contour of the pitch sequence.

OPTIONAL ARGUMENTS:

  - If the optional argument format is used, the first argument is to be
    a list of numbers representing the general contour of the pitch sequence,
    and the second is to be a symbol for the pitch-seq object's ID.

RETURN VALUE:

 - A pitch-seq object.

EXAMPLE:

```
;; The first creation option is using one argument that is a two-item list,
;; whereby the first item is a symbol to be used as the pitch-seq object's ID
;; and the second is a list of numbers representing the general contour of the
;; pitch sequence.
(make-pitch-seq '(pseq1 (1 2 1 1 3)))

=>
PITCH-SEQ: notes: NIL
highest: 3
lowest: 1
original-data: (1 2 1 1 3)
user-id: T
instruments: NIL
relative-notes: (not printed for sake of brevity)
relative-notes-length: 25
SCLIST: sclist-length: 5, bounds-alert: T, copy: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: PSEQ1, tag: NIL,
data: (1 2 1 1 3)

;; The second creation option uses two arguments, the first of which is a list
;; of numbers representing the general contour of the pitch sequence, the
;; second of which is a symbol which will be used as the pith-seq object's ID.
(make-pitch-seq '(2 1 1 3 1) 'pseq2)

=>
PITCH-SEQ: notes: NIL
highest: 3
lowest: 1
original-data: (2 1 1 3 1)
user-id: NIL
instruments: NIL
relative-notes: (not printed for sake of brevity)
relative-notes-length: 25
SCLIST: sclist-length: 5, bounds-alert: T, copy: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
```

```
NAMED-OBJECT: id: PSEQ2, tag: NIL,
data: (2 1 1 3 1)
```

```
;; An example assigning a pitch-seq only to specific instruments: ; ;
(make-pitch-seq '((1 2 1 1 3) violin flute) 'ps1))
```

SYNOPSIS:

```
(defun make-pitch-seq (id-data &optional (id nil))
```

28.5.471   sclist/plot

[sclist] [Methods]

DESCRIPTION:

Create text and data files suitable for plotting the numbers in the data (or
other) slot with gnuplot. The file name should be given without extension,
as the method will create a .txt and a .data file, for the command and data
files respectively. It is assumed that the data to be plotted is a list of
numbers. An error will be triggered if not.

The user must then call gnuplot in a terminal, in a manner such as "gnuplot
sclist.txt; open sclist.ps".

The method will create files that draw data points connected by lines by
default.

ARGUMENTS:

- An sclist object.
- A string that is the directory path and base file name (without
  extension) of the files to create.

OPTIONAL ARGUMENTS:

- The slot to plot. By default this will be the data slot, but derived
  classes may want to plot a different list slot, e.g. the kernels slot of
  the popcorn class.
- T or NIL to indicate whether to connect points by lines. T = draw
  lines. Default = T.

RETURN VALUE:

Returns the number of points plotted.

EXAMPLE:

```
(let ((ppcn (make-popcorn '(0.01 0.02) :min-spike 3.0 :max-spike 5.0)))
  (fit-to-length ppcn 100)
  (plot ppcn "/tmp/ppcn" 'kernels))

then in a terminal:
gnuplot ppcn.txt

this will create the postscript file ppcn.ps
```

SYNOPSIS:

```
(defmethod plot ((scl sclist) file &optional (slot 'data) (lines t))
```

28.5.472   sclist/rthm-seq

[sclist] [Classes]

NAME:

```
 rthm-seq

 File:              rthm-seq.lsp

 Class Hierarchy:  named-object -> linked-named-object -> sclist -> rthm-seq

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of the rthm-seq class which holds the bars
                   and rhythms of a sequence (multiple bars).  This will
                   generally be stored in a rthm-seq-palette and referenced
                   later in the rthm-seq-map.

                   The data used to create such an object will look
                   something like:

                   (rthm1 ((((2 4) q (q))
                            (s x 4 (e) e)
                            ((3 8) (e) e (e)))
                          :pitch-seq-palette '((psp1 (1 2 1 2 3 2 1))
                                               (psp2 (3 2 4 6 1 5 7))
                                               (psp3 (2 3 4 1 3 4 5)))))
```

```
 Author:            Michael Edwards: m@michael-edwards.org

 Creation date:    14th February 2001

 $$ Last modified:  17:08:54 Sat Jun  6 2020 CEST

 SVN ID: $Id$
```

28.5.473   rthm-seq/add-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Add a rthm-seq-bar object to the end of a given rthm-seq object.

 NB: If the rthm-seq-bar object is added without specifying a
     pitch-seq-palette, the method automatically adds data to the existing
     pitch-seq-palette.

ARGUMENTS:

 - A rhtm-seq object.
 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - A pitch-seq-palette.

RETURN VALUE:

 Returns the new value of the DURATION slot of the given rthm-seq object.

EXAMPLE:

```
;; Returns the new value of the DURATION slot
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (add-bar rs (make-rthm-seq-bar '((5 8) e e+32 s. +q))))

=> 10.5
```

```
;; Apply the method and print the rhythms objects of the given rthm-seq object
;; to see the changes
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (add-bar rs (make-rthm-seq-bar '((5 8) e e+32 s. +q)))
  (print-simple rs))

=>
rthm-seq NIL
(2 4): note Q, note E, note S, note S,
(2 4): rest E, note Q, rest E,
(3 8): note S, note S, note E., note S,
(5 8): note E, note E, note 32, note S., note Q,

;; Apply the method and print the DATA slot of the updated PITCH-SEQ-PALETTE
;; slot to see the new notes that have been automatically added
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (add-bar rs (make-rthm-seq-bar '((5 8) e e+32 s. +q)))
  (data (first (data (pitch-seq-palette rs)))))

=> (1 2 3 1 1 2 3 4 3 4 3)
```

SYNOPSIS:

```
(defmethod add-bar ((rs rthm-seq) (rsb rthm-seq-bar) &optional psp)
```

28.5.474   rthm-seq/change-pitches

[rthm-seq] [Methods]

DATE:

 February 3rd 2018, Heidhausen

DESCRIPTION:

 Change the pitches in a rthm-seq. The replacement pitches list can contain
 fewer pitches than in the seq: the method stops when it runs out. If there
 are more pitches than necessary, a list of unused pitches will be returned.

ARGUMENTS:

 - the rthm-seq object
 - a list of pitch objects or symbols as replacements
 - the bar within the sequence to start at (counting from 1)
 - the event within the start bar to start at (counting from 1)

OPTIONAL ARGUMENTS:

 keyword arguments:
 :written: whether to change the written or sounding pitches. Default = NIL =
 sounding pitches.

RETURN VALUE:

 if all pitches given are used, then NIL, otherwise the unused pitches as a
 list.

SYNOPSIS:

```
(defmethod change-pitches ((rs rthm-seq)
                           pitch-list
                           (start-bar integer) ; don't use bar-holder method
                           start-event
                           &key written)
```

28.5.475   rthm-seq/chop

[rthm-seq] [Methods]

DESCRIPTION:

 Applies the chop method to each rthm-seq-bar object contained in the given
 rthm-seq object (see rthm-seq-bar::chop for details), returning a list of
 rthm-seq objects, each of which contains just one of the rthm-seq-bar
 objects created with chop.

 The chop method is the basis for slippery-chicken's feature of
 intra-phrasal looping.

 NB: Since the chop method functions by comparing each beat of a given
     rthm-seq-bar object to the specified <chop-points> pattern for
     segmenting that beat, all rthm-seq-bar objects in the given rthm-seq
     object must be evenly divisible by the beat for which the pattern is
     defined. For example, if the <chop-points> argument defines a
     quarter-note, all bars in the given rthm-seq object must be evenly
     divisible by a quarter-note, and a rthm-seq object consisting of a 2/4,

a 3/4 and a 3/8 bar would fail at the 3/8 bar with an error.

NB: The <unit> argument must be a duplet rhythmic value (i.e. 32, 's, 'e
    etc.) and cannot be a tuplet value (i.e. 'te 'fe etc.).

NB: In order for the resulting chopped rhythms to be parsable by LilyPond
    and CMN, there can be no tuplets (triplets etc.) among the rhythms to
    be chopped. Such rhythms will result in LilyPond and CMN errors. This
    has only minimal bearing on any MIDI files produced, however, and these
    can potentially be imported into notation software.

ARGUMENTS:

 - A rthm-seq object.

OPTIONAL ARGUMENTS:

 - <chop-points>. A list of integer pairs, each of which delineates a
   segment of the beat of the given rthm-seq-bar object measured in the
   rhythmic unit specified by the <unit> argument. See the documentation for
   rthm-seq-bar::chop for more details.
 - <unit>. The rhythmic duration that serves as the unit of measurement for
   the chop points. Default = 's.
 - <number-bars-first>. T or NIL. This argument helps in naming (and
   therefore debugging) the newly-created bars. If T, the bars in the
   original rthm-seq will be renumbered, starting from 1, and this will be
   reflected in the tag of the new bars. E.g. if T, a new bar's tag may be
   new-bar-from-rs1-b3-time-range-1.750-to-2.000, if NIL this would be
   new-bar-from-rs1-time-range-1.750-to-2.000. Default = T.

RETURN VALUE:

 A list of rthm-seq objects.

EXAMPLE:

```
;; Create a rthm-seq with three bars, all having a quarter-note beat basis,
;; apply chop, and print-simple the resulting list of new rthm-seq-bar
;; objects. The rthm-seq numbers printed with this are the IDs of the rthm-seq
;; objects, not the bar-nums of the individual rthm-seq-bar objects.
(let* ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s)
                                   ((e) q (e))
                                   (s s (e) e. s))
                                  :pitch-seq-palette ((1 2 3 4 5 6 7 8 9)
                                                      (9 8 7 6 5 4 3 2 1)))))))
```

```
      (ch (chop rs
                '((1 1) (1 2) (1 3) (1 4) (2 2) (2 3) (2 4) (3 3) (3 4) (4 4))
                's)))
  (loop for rs-obj in ch do (print-simple rs-obj)))

=>
rthm-seq 1
(1 16): NIL S,
rthm-seq 2
(1 8): NIL E,
rthm-seq 3
(3 16): NIL E.,
rthm-seq 4
(1 4): NIL Q,
rthm-seq 5
(1 16): rest 16,
rthm-seq 6
(1 8): rest 8,
rthm-seq 7
(3 16): rest 16/3,
rthm-seq 8
(1 16): rest 16,
rthm-seq 9
(1 8): rest 8,
rthm-seq 10
(1 16): rest 16,
rthm-seq 11
(1 16): NIL S,
rthm-seq 12
(1 8): NIL E,
rthm-seq 13
(3 16): NIL E, NIL S,
rthm-seq 14
(1 4): NIL E, NIL S, NIL S,
rthm-seq 15
(1 16): rest 16,
rthm-seq 16
(1 8): rest S, NIL S,
rthm-seq 17
(3 16): rest S, NIL S, NIL S,
rthm-seq 18
(1 16): NIL S,
rthm-seq 19
(1 8): NIL S, NIL S,
rthm-seq 20
(1 16): NIL S,
```

```
rthm-seq 21
(1 16): rest 16,
rthm-seq 22
(1 8): rest 8,
rthm-seq 23
(3 16): rest E, NIL S,
rthm-seq 24
(1 4): rest E, NIL E,
rthm-seq 25
(1 16): rest 16,
rthm-seq 26
(1 8): rest S, NIL S,
rthm-seq 27
(3 16): rest S, NIL E,
rthm-seq 28
(1 16): NIL S,
rthm-seq 29
(1 8): NIL E,
rthm-seq 30
(1 16): rest 16,
rthm-seq 31
(1 16): rest 16,
rthm-seq 32
(1 8): rest 8,
rthm-seq 33
(3 16): rest 16/3,
rthm-seq 34
(1 4): rest 4,
rthm-seq 35
(1 16): rest 16,
rthm-seq 36
(1 8): rest 8,
rthm-seq 37
(3 16): rest 16/3,
rthm-seq 38
(1 16): rest 16,
rthm-seq 39
(1 8): rest 8,
rthm-seq 40
(1 16): rest 16,
rthm-seq 41
(1 16): NIL S,
rthm-seq 42
(1 8): NIL S, NIL S,
rthm-seq 43
(3 16): NIL S, NIL S, rest S,
```

```
rthm-seq 44
(1 4): NIL S, NIL S, rest E,
rthm-seq 45
(1 16): NIL S,
rthm-seq 46
(1 8): NIL S, rest S,
rthm-seq 47
(3 16): NIL S, rest E,
rthm-seq 48
(1 16): rest 16,
rthm-seq 49
(1 8): rest 8,
rthm-seq 50
(1 16): rest 16,
rthm-seq 51
(1 16): NIL S,
rthm-seq 52
(1 8): NIL E,
rthm-seq 53
(3 16): NIL E.,
rthm-seq 54
(1 4): NIL E., NIL S,
rthm-seq 55
(1 16): rest 16,
rthm-seq 56
(1 8): rest 8,
rthm-seq 57
(3 16): rest E, NIL S,
rthm-seq 58
(1 16): rest 16,
rthm-seq 59
(1 8): rest S, NIL S,
rthm-seq 60
(1 16): NIL S,

;; Attempting to apply the method to a rthm-seq object in which not all bars
;; have time-signatures that are divisible by the beat defined in the
;; <chop-points> argument will result in dropping into the debugger with an
;; error
(let* ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s)
                                   ((e) q (e))
                                   ((3 8) (e) e. s))
                                  :pitch-seq-palette ((1 2 3 4 5 6 7)
                                                      (9 8 7 6 5 4 3)))))))
       (ch (chop rs
                 '((1 1) (1 2) (1 3) (1 4) (2 2) (2 3) (2 4) (3 3) (3 4) (4 4))
```

```
                  's)))
  (loop for rs-obj in ch do (print-simple rs-obj)))

=>
rthm-seq-bar::get-beats: Can't find an exact beat of rhythms
   (dur: 0.75 beat-dur: 0.5)!
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod chop ((rs rthm-seq) &optional chop-points (unit 's)
                 (number-bars-first t))
```

28.5.476   rthm-seq/clear-ties-beg-end

[rthm-seq] [Methods]

DESCRIPTION:

 Deletes ties to the first rhythm(s) of a rthm-seq, making them rests
 instead of tied notes. Also makes sure the last rhythm is not tied from
 (into another sequence). Useful if you're making rthm-seqs from other
 (larger) rthm-seqs or algorithmically.

ARGUMENTS:

 - the rthm-seq object

RETURN VALUE:

 T if any ties were cleared, NIL otherwise.

SYNOPSIS:

```
(defmethod clear-ties-beg-end ((rs rthm-seq))
```

28.5.477   rthm-seq/combine

[rthm-seq] [Methods]

DESCRIPTION:

 Combine two rthm-seqs into one, updating slots for the new object, which is
 a clone.

NB: The MARKS slot is ignored for now (it is as of yet

NB: This method sets the values of the individual slots but leaves the DATA
    slot untouched (for cases in which the user might want to see where the
    new data originated from, or otherwise use the old data somehow, such
    as in a new rthm-seq object).

ARGUMENTS:

 - A first rthm-seq object.
 - A second rthm-seq object.

RETURN VALUE:

 - A rthm-seq object.

EXAMPLE:

```
;; The method returns a rthm-seq object
(let ((rs1 (make-rthm-seq '((((2 4) q+e s s)
                             ((e) q (e))
                             ((3 8) s s e. s))
                            :pitch-seq-palette ((1 2 3 1 1 2 3 4)))))
      (rs2 (make-rthm-seq '((((4 4) h+e (e) { 3 te te te })
                             ((5 8) e e+32 s. +q)
                             ((3 4) (q) q q))
                            :pitch-seq-palette ((1 2 3 4 1 2 3 1 2))))))
  (combine rs1 rs2))

=>
RTHM-SEQ: num-bars: 6
          num-rhythms: 25
          num-notes: 17
          num-score-notes: 21
          num-rests: 4
          duration: 15.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
          (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 6, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "NIL-NIL", tag: NIL,
```

```
data: ((((2 4) Q+E S S) ((E) Q (E)) ((3 8) S S E. S)) PITCH-SEQ-PALETTE
       ((1 2 3 1 1 2 3 4))
       (((4 4) H+E (E) { 3 TE TE TE }) ((5 8) E E+32 S. +Q) ((3 4) (Q) Q Q))
       PITCH-SEQ-PALETTE ((1 2 3 4 1 2 3 1 2)))

;; With the same combine call, print the collected contents of the BARS slot
;; and the PITCH-SEQ-PALETTE slot of the new rthm-seq object
(let* ((rs1 (make-rthm-seq '((((2 4) q+e s s)
                              ((e) q (e))
                              ((3 8) s s e. s))
                             :pitch-seq-palette ((1 2 3 1 1 2 3 4)))))
       (rs2 (make-rthm-seq '((((4 4) h+e (e) { 3 te te te })
                              ((5 8) e e+32 s. +q)
                              ((3 4) (q) q q))
                             :pitch-seq-palette ((1 2 3 4 1 2 3 1 2)))))
       (crs (combine rs1 rs2)))
  (print-simple crs)
  (print (data (get-first (pitch-seq-palette crs)))))

=>
rthm-seq NIL-NIL
(2 4): note Q, note E, note S, note S,
(2 4): rest E, note Q, rest E,
(3 8): note S, note S, note E., note S,
(4 4): note H, note E, rest E, note TE, note TE, note TE,
(5 8): note E, note E, note 32, note S., note Q,
(3 4): rest 4, note Q, note Q,
(1 2 3 1 1 2 3 4 1 2 3 4 1 2 3 1 2)
```

SYNOPSIS:

```
(defmethod combine ((rs1 rthm-seq) (rs2 rthm-seq))
```

28.5.478   rthm-seq/delete-marks

[rthm-seq] [Methods]

DESCRIPTION:

 Delete all marks from the MARKS slot of the specified rthm-seq object and
 replace them with NIL.

ARGUMENTS:

 - A rthm-seq object

RETURN VALUE:

 NIL

EXAMPLE:

```
(let ((mrs (make-rthm-seq '(seq1 ((((2 4) q e (s) s))
                                   :pitch-seq-palette ((1 2 3))
                                   :marks (ff 1 a 1 pizz 1 ppp 2 s 2))))))
  (print (marks mrs))
  (delete-marks mrs)
  (print (marks mrs)))

=>
((FF 1) (A 1) (PIZZ 1) (PPP 2) (S 2))
NIL
```

SYNOPSIS:

```
(defmethod delete-marks ((rs rthm-seq))
```

28.5.479   rthm-seq/get-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Get a specified rthm-seq-bar object from within a rthm-seq object.

ARGUMENTS:

 - A rthm-seq object.
 - An integer that is the 1-based number of the desired bar to return from
   within the given rthm-seq object.

RETURN VALUE:

 Returns a rthm-seq-bar object.

EXAMPLE:

```
;;; Returns a rthm-seq-bar object
(let ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s)
                                  ((e) q (e))
```

```
                                      ((3 8) s s e. s)))))))
  (get-bar rs 2))

=>
RTHM-SEQ-BAR: time-sig: 0 (2 4), time-sig-given: NIL, bar-num: -1,
              old-bar-nums: NIL, write-bar-num: NIL, start-time: -1.000,
              start-time-qtrs: -1.0, is-rest-bar: NIL, multi-bar-rest: NIL,
              show-rest: T, notes-needed: 1,
              tuplets: NIL, nudge-factor: 0.35, beams: NIL,
              current-time-sig: 6, write-time-sig: NIL, num-rests: 2,
              num-rhythms: 3, num-score-notes: 1, parent-start-end: NIL,
              missing-duration: NIL, bar-line-type: 0,
              player-section-ref: NIL, nth-seq: NIL, nth-bar: NIL,
              rehearsal-letter: NIL, all-time-sigs: (too long to print)
              sounding-duration: NIL,
              rhythms: (
[...]

(let ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s)
                                  ((e) q (e))
                                  ((3 8) s s e. s)))))))
  (print-simple (get-bar rs 2)))

=> (2 4): rest E, note Q, rest E,
```

SYNOPSIS:

```
(defmethod get-bar ((rs rthm-seq) bar-num &optional ignore)
```

28.5.480   rthm-seq/get-first

[rthm-seq] [Methods]

DESCRIPTION:

 Return the first rhythm/event object in the specified rthm-seq object.

ARGUMENTS:

 - A rthm-seq object.

RETURN VALUE:

 A rhythm/event object.

EXAMPLE:

```
(let ((mrs (make-rthm-seq '(seq1 ((((2 4) q e (s) s))
                                  :pitch-seq-palette ((1 2 3))
                                  :marks (ff 1 a 1 pizz 1 ppp 2 s 2))))))
  (get-first mrs))

=>
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
        is-whole-bar-rest: NIL,
        score-rthm: 4.0, undotted-value: 4, num-flags: 0, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: (PIZZ A FF), marks-in-part: NIL,
        letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q
```

SYNOPSIS:

```
(defmethod get-first ((rs rthm-seq))
```

28.5.481   rthm-seq/get-last

[rthm-seq] [Methods]

DESCRIPTION:

 Return the last rhythm/event object in the specified rthm-seq object.

ARGUMENTS:

 - A rthm-seq object.

RETURN VALUE:

 A rhythm/event object.

EXAMPLE:

```
(let ((mrs (make-rthm-seq '(seq1 ((((2 4) q e (s) s))
                                  :pitch-seq-palette ((1 2 3))
                                  :marks (ff 1 a 1 pizz 1 ppp 2 s 2))))))
  (get-last mrs))

=>
```

```
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
        is-whole-bar-rest: NIL,
        score-rthm: 16.0, undotted-value: 16, num-flags: 2, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.250,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 16, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: S, tag: NIL,
data: S
```

SYNOPSIS:

```
(defmethod get-last ((rs rthm-seq))
```

28.5.482   rthm-seq/get-last-attack

[rthm-seq] [Methods]

DESCRIPTION:

 Gets the rhythm object for the last note that needs an attack (i.e. not a
 rest and not a tied note) in a given rthm-seq object.

ARGUMENTS:

 - A rthm-seq object.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print a warning message if the given index
   (minus 1) is greater than the number of attacks in the rthm-seq object
   (default = T). This is a carry-over argument from the get-nth-attack
   method called within the get-last-attack method and not likely to be
   needed for use with get-last-attack.

RETURN VALUE:

 A rhythm object.

EXAMPLE:

```
;; Returns a rhythm object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
```

```
                                    ((e) q (e))
                                    ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-last-attack rs))

=>
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
        score-rthm: 16.0f0, undotted-value: 16, num-flags: 2, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.250,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 16, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: S, tag: NIL,
data: S
```

SYNOPSIS:

```
(defmethod get-last-attack ((rs rthm-seq) &optional (warn t))
```

28.5.483   rthm-seq/get-last-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Get the last rthm-seq-bar object of a given rthm-seq object.

ARGUMENTS:

 - A rthm-seq object.

RETURN VALUE:

 A rthm-seq-bar object.

EXAMPLE:

```
;;; The method returns a rthm-seq-bar object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-last-bar rs))
```

```
=>
RTHM-SEQ-BAR: time-sig: 6 (3 8), time-sig-given: T, bar-num: -1,
[...]
data: ((3 8) S S E. S)
```

SYNOPSIS:

```
(defmethod get-last-bar ((rs rthm-seq))
```

28.5.484   rthm-seq/get-last-event

[rthm-seq] [Methods]

DESCRIPTION:

 Get the last event object (or rhythm object) of a given rthm-seq-bar
 object.

ARGUMENTS:

 - A rthm-seq object.

RETURN VALUE:

 Returns an event (or rhythm) object.

EXAMPLE:

```
;; The last event is a rhythm object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-last-event rs))

=>
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
        score-rthm: 16.0f0, undotted-value: 16, num-flags: 2, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.250,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 16, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: S, tag: NIL,
data: S
```

```
;; The last event is an event object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. ,(make-event 'c4 's)))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-last-event rs))

=>
EVENT: start-time: NIL, end-time: NIL,
[...]
PITCH: frequency: 261.6255569458008, midi-note: 60, midi-channel: NIL
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
NAMED-OBJECT: id: S, tag: NIL,
data: S
```

SYNOPSIS:

```
(defmethod get-last-event ((rs rthm-seq))
```

28.5.485   rthm-seq/get-multipliers

[rthm-seq] [Methods]

DESCRIPTION:

Get a list of factors by which a specified rhythmic unit must be multiplied
in order to create the rhythms of a given rthm-seq object.

NB: The get-multipliers method determines durations in the source rhythmic
    material based on attacked notes only, so beginning ties will be
    ignored and rests following an attack will count the same as if the
    attacked note were tied to another note with the same duration as the
    rest. For this reason, the results returned by the method when applied
    to a rthm-seq object that contains multiple bars may differ from
    applying the method to multiple rthm-seqs with single bars, albeit with
    the same rhythms when seen as a group (see example below).

ARGUMENTS:

 - A rthm-seq object.
 - A rhythm unit, either as a number or a shorthand symbol (i.e. 's)

OPTIONAL ARGUMENTS:

  – T or NIL to indicate whether to round the results. T = round.
    Default = NIL. NB: Lisp always rounds to even numbers, meaning x.5 may
    sometimes round up and sometimes round down; thus (round 1.5) => 2, and
    (round 2.5) => 2.

RETURN VALUE:

 A list of numbers.

EXAMPLE:

```
;;; By default the method returns the list of multipliers un-rounded
(let ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))))
  (get-multipliers rs 'e))

=> (2.0 1.0 0.5 0.5)

;; Setting the optional argument to T rounds the results before returning
(let ((rs (make-rthm-seq '(seq1 ((((2 4) q e s s))
                                :pitch-seq-palette ((1 2 3 4)))))))
  (get-multipliers rs 'e t))

=> (2 1 0 0)

;;; Applying the method to the a multiple-bar rthm-seq object may return
;;; different results than applying the method to each of the bars contained
;;; within that rthm-seq object as individual one-bar rthm-seq objects, as the
;;; method measures the distances between attacked notes, regardless of ties
;;; and rests.
(let ((rs1 (make-rthm-seq '(seq1 ((((2 4) q +e. s))
                                  :pitch-seq-palette ((1 2))))))
      (rs2 (make-rthm-seq '(seq2 ((((2 4) (s) e (s) q))
                                  :pitch-seq-palette ((1 2))))))
      (rs3 (make-rthm-seq '(seq3 ((((2 4) +e. s { 3 (te) te te } ))
                                  :pitch-seq-palette ((1 2 3))))))
      (rs4 (make-rthm-seq '(seq4 ((((2 4) q +e. s)
                                   ((s) e (s) q)
                                   (+e. s { 3 (te) te te } ))
                                  :pitch-seq-palette ((1 2 3 4 5 6 7)))))))
  (print (get-multipliers rs1 'e))
  (print (get-multipliers rs2 'e))
  (print (get-multipliers rs3 'e))
  (print (get-multipliers rs4 'e)))

=>
```

```
(3.5 0.5)
(1.5 2.0)
(1.1666666666666665 0.6666666666666666 0.6666666666666666)
(3.5 1.0 1.5 3.5 1.1666666666666665 0.6666666666666666 0.6666666666666666)
```

SYNOPSIS:

```
(defmethod get-multipliers ((rs rthm-seq) rthm &optional round ignore)
```

28.5.486   rthm-seq/get-nth-attack

[rthm-seq] [Methods]

DESCRIPTION:

Gets the rhythm object for the nth note in a given rthm-seq object that
needs an attack, i.e. not a rest and not tied.

ARGUMENTS:

- The zero-based index number indicating which attack is sought.
- The given rthm-seq object in which to search.

OPTIONAL ARGUMENTS:

- T or NIL indicating whether to print an error message if the given index
  is greater than the number of attacks (minus 1) in the rthm-seq object
  (default = T).

RETURN VALUE:

A rhythm object.

EXAMPLE:

```
;; The method returns a rhythm object when successful
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-attack 4 rs))

=>
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
```

```
        score-rthm: 16.0f0, undotted-value: 16, num-flags: 2, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.250,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 16, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: S, tag: NIL,
data: S


;; By default, the method drops into the debugger with an error when the
;; specified index is greater than the number of items in the given rthm-seq
;; object.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-attack 11 rs))

=>
rthm-seq::get-nth-attack: Couldn't get attack with index 11
   [Condition of type SIMPLE-ERROR]

;; This error can be suppressed, simply returning NIL, by setting the optional
;; argument to NIL.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-attack 11 rs nil))

=> NIL, 0, NIL
```

SYNOPSIS:

```
(defmethod get-nth-attack (index (rs rthm-seq)
                           &optional (error t))
```

28.5.487   rthm-seq/get-nth-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Get the nth rthm-seq-bar object from a given rthm-seq object.

ARGUMENTS:

- A rthm-seq object.
- An index number (zero-based).

RETURN VALUE:

Returns a rthm-seq-bar object if successful.

Returns NIL and prints a warning if the specified index number is greater
than the number of rthm-seq-bar objects (minus one) in the given rthm-seq
object.

EXAMPLE:

```
;;; The method returns a rhtm-seq-bar object when successful
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-bar 1 rs))

=>
RTHM-SEQ-BAR: time-sig: 0 (2 4), time-sig-given: NIL, bar-num: -1,
[...]
NAMED-OBJECT: id: "NIL-bar2", tag: NIL,
data: ((E) Q (E))

;; Returns a warning and prints NIL when the specified index number is greater
;; than the number of rthm-seq-bar objects in the given rthm-seq object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-bar 11 rs))

=> NIL
WARNING: rthm-seq::rthm-seq-check-bounds: Illegal list reference: 11
```

SYNOPSIS:

```
(defmethod get-nth-bar (nth (rs rthm-seq)))
```

28.5.488   rthm-seq/get-nth-non-rest-rhythm

[rthm-seq] [Methods]

DESCRIPTION:

Get the nth non-rest rhythm object stored in the given rthm-seq object.

ARGUMENTS:

 - The zero-based index number indicating which non-rest-rhythm is sought.
 - The given rthm-seq object in which to search.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print an error message if the given index
   is greater than the number of non-rest rhythms (minus 1) in given
   rthm-seq object. (Default = T.)

RETURN VALUE:

 A rhythm object.

 Returns NIL if the given index is higher than the highest possible index of
 non-rest rhythms in the given rthm-seq-bar object.

EXAMPLE:

```
;; The method returns a rhythm object when successful
(let ((rs (make-rthm-seq '((((2 4) q e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3 4))))))
  (get-nth-non-rest-rhythm 4 rs))

=>
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
        score-rthm: 4.0f0, undotted-value: 4, num-flags: 0, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q

;; By default, the method drops into the debugger with an error when the
;; specified index is greater than the number of items in the given rthm-seq
;; object.
(let ((rs (make-rthm-seq '((((2 4) q e s s)
                            ((e) q (e))
```

```
                              ((3 8) s s e. s))
                              :pitch-seq-palette ((1 2 3 4 1 1 2 3 4))))))
  (get-nth-non-rest-rhythm 11 rs))

=>
rthm-seq::get-nth-non-rest-rhythm: Couldn't get non-rest rhythm with index 11
   [Condition of type SIMPLE-ERROR]

;; This error can be suppressed, simply returning NIL, by setting the optional
;; argument to NIL.
(let ((rs (make-rthm-seq '((((2 4) q e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                            :pitch-seq-palette ((1 2 3 4 1 1 2 3 4))))))
  (get-nth-non-rest-rhythm 11 rs nil))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-non-rest-rhythm (index (rs rthm-seq)
                                     &optional (error t))
```

28.5.489   rthm-seq/get-nth-rhythm

[rthm-seq] [Methods]

DESCRIPTION:

 Gets the rhythm (or event) object for the nth note in a given rthm-seq
 object.

ARGUMENTS:

 - The zero-based index number indicating which attack is sought.
 - The given rthm-seq object in which to search.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print an error message if the given index
   is greater than the number of attacks (minus 1) in the rthm-seq object
   (default = T).

RETURN VALUE:

 A rhythm or event object.

EXAMPLE:

```
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (get-nth-rhythm 4 rs))

=>

RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: T,
        score-rthm: 8.0f0, undotted-value: 8, num-flags: 1, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.500,
        is-grace-note: NIL, needs-new-note: NIL, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 8, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E
**************
```

SYNOPSIS:

```
(defmethod get-nth-rhythm (index (rs rthm-seq) &optional (error t))
```

28.5.490   rthm-seq/get-rhythms

[rthm-seq] [Methods]

DESCRIPTION:

 Get the rhythm objects in a given rthm-seq object, contained in a list.

ARGUMENTS:

 - A rthm-seq object.

RETURN VALUE:

 A list.

EXAMPLE:

```
;; Returns a list of rhythm objects
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (get-rhythms rs))

=>
(
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
[...]
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: NIL,
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: T,
[...]
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
[...]
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: T,
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
RHYTHM: value: 5.333, duration: 0.750, rq: 3/4, is-rest: NIL,
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: NIL,
[...]
)

;; Get just the rhythm labels from the same rthm-seq object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (loop for r in (get-rhythms rs) collect (data r)))

=> ("Q" "E" S S E Q E S S E. S)
```

SYNOPSIS:

```
(defmethod get-rhythms ((rs rthm-seq))
```

28.5.491   rthm-seq/get-time-sigs

[rthm-seq] [Methods]

DESCRIPTION:

Return a list of time-sig objects for each of the rthm-seq-bar objects in a
given rthm-seq object.

One time signature is returned for each rthm-seq-bar object, even if two or
more consecutive objects have the same time signature.

Optionally, this method can return a list of time signatures in list form
(e.g. ((2 4) (3 4)) etc.) rather than a list of time-sig objects.

ARGUMENTS:

 - A rthm-seq object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to return the time signatures as time-sig
   objects or a list of two-item lists. T = time-sig objects. Default = T.

RETURN VALUE:

Returns a list of time-sig objects by default. Optionally a list of time
signatures as two-item lists can be returned instead.

EXAMPLE:

```
;; Return a list of time-sig objects, one for each rthm-seq-bar object even if
;; consecutive rthm-seq-bar objects have the same time signature
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (get-time-sigs rs))

=> (
TIME-SIG: num: 2, denom: 4, duration: 2.0, compound: NIL, midi-clocks: 24, num-beats: 2
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0204", tag: NIL,
data: (2 4)
```

```
**************


TIME-SIG: num: 2, denom: 4, duration: 2.0, compound: NIL, midi-clocks: 24, num-beats: 2
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0204", tag: NIL,
data: (2 4)
**************


TIME-SIG: num: 3, denom: 8, duration: 1.5, compound: T, midi-clocks: 24, num-beats: 1
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0308", tag: NIL,
data: (3 8)
**************
)

;; Return the same as a list of two-item lists instead
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (get-time-sigs rs t))

=> ((2 4) (2 4) (3 8))
```

SYNOPSIS:

```
(defmethod get-time-sigs ((rs rthm-seq) &optional as-list)
```

28.5.492   rthm-seq/insert-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Insert a rthm-seq-bar object into the given rthm-seq object and
 re-initialize it. If there's a pitch-seq/pitch-seq-palette given (list of
 numbers, or list of lists), splice this in at the appropriate location.

 NB: This method sets the values of the individual slots but leaves the DATA
     slot untouched (for cases in which the user might want to see where the
     new data originated from, or otherwise use the old data somehow, such
     as in a new rthm-seq object).

ARGUMENTS:

 - A rthm-seq object.
 - A rthm-seq-bar object.
 - A bar number (integer). This argument is the bar number of the bar to be
   inserted, relative to the rthm-seq and 1-based; e.g., if 3, then it will
   come before the present third bar.

OPTIONAL ARGUMENTS:

 - A pitch-seq object.
 - (three ignore arguments for sc-internal use only)

RETURN VALUE:

 Returns T if successful.

 Drops into the debugger with an error if the specified bar-number argument
 is greater than the number of rthm-seq-bar objects in the given rthm-seq.

EXAMPLE:

```
;; The method returns T when successful
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (insert-bar rs (make-rthm-seq-bar '((3 4) q. e e s s)) 3))

=> T

;; Create a rthm-seq object with three rthm-seq-bars and print the contents of
;; the NUM-BARS slot to confirm that it contains 3 objects. Insert a bar before
;; the third item and print the value of the NUM-BARS slot again to confirm
;; that there are now 4 objects. Use print-simple and get-nth-bar to confirm
;; that the 3rd object (with a zero-based index of 2) is indeed the one
;; inserted.

(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (print (num-bars rs))
  (insert-bar rs (make-rthm-seq-bar '((3 4) q. e e s s)) 3)
  (print (num-bars rs))
```

```
  (print-simple (get-nth-bar 2 rs)))

=>
3
4
(3 4): note Q., note E, note E, note S, note S,

;; Attempting to insert a bar with an index number greater than the number of
;; objects currently in the rthm-seq object drops into the debugger with an
;; error
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (insert-bar rs (make-rthm-seq-bar '((3 4) q. e e s s)) 11))

=>
rthm-seq::insert-bar: only 3 bars in rthm-seq!
   [Condition of type SIMPLE-ERROR]

;; Inserting a rthm-seq-bar using the optional pitch-seq argument splices the
;; specified value of that argument into the existing pitch-seq-palette
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (insert-bar rs (make-rthm-seq-bar '((3 4) q. e e s s)) 3 '((1 2 3 4 5)))
  (data (get-first (pitch-seq-palette rs))))

=> (1 2 3 1 1 2 3 4 5 1 2 3 4)
```

SYNOPSIS:

```
(defmethod insert-bar ((rs rthm-seq) (rsb rthm-seq-bar) bar-num
                        &optional pitch-seq ignore1 ignore2 ignore3)
```

28.5.493   rthm-seq/make-rhythms

[rthm-seq] [Functions]

DATE:

 11 Feb 2010

DESCRIPTION:

Initialize a group of rhythms, taking advantage of rthm-seq's ability to
add tuplet and beaming info.

ARGUMENTS:

 - A list of rhythms equalling one full bar
 - The time signature of that bar as a list (e.g (2 4))

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to divide the resulting list into sublists,
   each of which are the equivalent of one beat long. Default = NIL.

RETURN VALUE:

 - A list

EXAMPLE:

```
;; Apply the function and test that the result is a list
(let ((rs (make-rhythms '(q e s s) '(2 4))))
  (listp rs))

=> T

;; Apply the function and see that we've created a list with 4 elements
(let ((rs (make-rhythms '(q e s s) '(2 4))))
  (length rs))

=> 4

;; Apply the function with the optional split-into-beats argument set to T and
;; see that we now have two lists, each equalling one beat in combined
;; length. Print the data of the contents.
(let ((rs (make-rhythms '(q e s s) '(2 4) t)))
  (print (length rs))
  (print (loop for b in rs collect (length b)))
  (print (loop for b in rs
            collect (loop for r in b
                       collect (data r)))))

=>
2
(1 3)
((Q) (E S S))
```

```
;; Apply the function using beam indications then print the BEAM slots of the
;; individual rhythm objects contained in the result
(let ((rs (make-rhythms '(q - e s s -) '(2 4))))
  (loop for r in rs collect (beam r)))

=> (NIL 1 NIL 0)

;; Apply the function using tuplet indications then print the BRACKET slots of
;; the individual rhythms objects contained in the result
(let ((rs (make-rhythms '( { 3 te te te } - e s s -) '(2 4))))
  (loop for r in rs collect (bracket r)))

=> (((1 3)) (-1) (1) NIL NIL NIL)
```

SYNOPSIS:

```
(defun make-rhythms (bar time-sig &optional split-into-beats)
```

28.5.494   rthm-seq/make-rthm-seq

[rthm-seq] [Functions]

DESCRIPTION:

Creates a rthm-seq object from a list of at least bars and generally also a
list of pitch sequences.

ARGUMENTS:

- A list with the following items:
  - A symbol that will be used as the ID of the seq
  - Another list, containing two items:
    - A list of rthm-seq-bars and
    - A list of pitch-seqs attached to the :pitch-seq-palette accessor

OPTIONAL ARGUMENTS:

keyword argument
- :psp-inversions. T or NIL to indicate whether to also automatically
  generate and add inverted forms of the specified pitch-seq objects.
  T = generate and add. Default = NIL.
- :warn-time-sigs. T or NIL to indicate whether to issue a warning when a
  time-signature is given where it's not needed (i.e. two bars of 4/4)

RETURN VALUE:

 Returns a rthm-seq object.

EXAMPLE:

```
;; Make a rthm-seq object with the ID seq1 that contains one 2/4 bar of
;; rhythms and one pitch sequence in the pitch-seq-palette
(make-rthm-seq '(seq1 ((((2 4) q e s s))
                       :pitch-seq-palette ((1 2 3 4)))))

=>
RTHM-SEQ: num-bars: 1
          num-rhythms: 4
          num-notes: 4
          num-score-notes: 4
          num-rests: 0
          duration: 2.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SEQ1, tag: NIL,
data: ((((2 4) Q E S S)) PITCH-SEQ-PALETTE (1 2 3 4))

;; A rthm-seq object with two bars of rhythms and two pitch-seqs in the
;; pitch-seq-palette. There must be as many items in each pitch-seq list as
;; there are rhythms in each rthm-seq-bar.
(make-rthm-seq '(seq1 ((((2 4) q e s s)
                         ((e) q (e)))
                       :pitch-seq-palette ((1 2 3 4 5)
                                           (2 4 6 8 10)))))

=>
RTHM-SEQ: num-bars: 2
          num-rhythms: 7
          num-notes: 5
          num-score-notes: 5
          num-rests: 2
          duration: 4.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
```

```
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SEQ1, tag: NIL,
data: ((((2 4) Q E S S) ((E) Q (E))) PITCH-SEQ-PALETTE
        ((1 2 3 4 5) (2 4 6 8 10)))

;; The pitch-seq-palette may be omitted, and time signatures may be changed
(make-rthm-seq '(seq1 ((((2 4) q e s s)
                        ((e) q (e))
                        ((3 8) s s e. s)))))

=>
RTHM-SEQ: num-bars: 3
          num-rhythms: 11
          num-notes: 9
          num-score-notes: 9
          num-rests: 2
          duration: 5.5
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 1, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SEQ1, tag: NIL,
data: ((((2 4) Q E S S) ((E) Q (E)) ((3 8) S S E. S)))

;;; With :psp-inversions set to T, the inverted forms of the specified
;;; pitch-seq are automatically generated and added
(let ((mrs
       (make-rthm-seq '(seq1 ((((2 4) q e s s))
                               :pitch-seq-palette ((1 2 3 4))))
                      :psp-inversions t)))
  (data (pitch-seq-palette mrs)))

=> (
PITCH-SEQ: notes: NIL
           highest: 4
           lowest: 1
           original-data: (1 2 3 4)
           user-id: NIL
           instruments: NIL
           relative-notes: (not printed for sake of brevity)
           relative-notes-length: 25
SCLIST: sclist-length: 4, bounds-alert: T, copy: NIL
```

```
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "rthm-seq-SEQ1-pitch-seq-palette-ps-1", tag: NIL,
data: (1 2 3 4)
**************


PITCH-SEQ: notes: NIL
           highest: 4
           lowest: 1
           original-data: (4 3 2 1)
           user-id: NIL
           instruments: NIL
           relative-notes: (not printed for sake of brevity)
           relative-notes-length: 25
SCLIST: sclist-length: 4, bounds-alert: T, copy: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "rthm-seq-SEQ1-pitch-seq-palette-ps-1-inverted", tag: NIL,
data: (4 3 2 1)
**************
)
```

SYNOPSIS:

```
(defun make-rthm-seq (rs &key (psp-inversions nil) (warn-time-sigs t))
```

28.5.495   rthm-seq/make-rthm-seq-from-fragments

[rthm-seq] [Functions]

DATE:

 January 2010

DESCRIPTION:

 Make a rthm-seq object from a predefined list of rhythm fragments.

 NB: No pitch-seqs can be passed as yet.

ARGUMENTS:

 - The ID of the rthm-seq object to be made.
 - A list of rhythm lists (fragments) paired with key IDs. The rhythm lists
   take the form of rthm-seq-bar definitions without the time signatures.
 - A list of lists containing any combination of the key IDs from the list

of fragments. These will be collated to create the resulting rthm-seq
object. Each element will make up one whole bar.
- A list of meters. These can be given either as single numerators, whereby
 the optional <default-beat> argument will then be the denominator) or
 two-item lists consisting of (num denom). There must be one meter for each
 item in the list of references, and the meters must correspond to the
 number of beats in the corresponding item from the list of references.

RETURN VALUE:

 A rthm-seq object.

EXAMPLE:

```
(let ((frags '((1 (- s s - (e)))
               (2 (s (s) (s) s))
               (3 ((s) - s e -))
               (4 (- s s (s) s -))
               (5 ((e) - s s -))
               (6 ((q))))))
  (make-rthm-seq-from-fragments
   'test-rs frags
   '((1 2 3) (1 4) (6 1) (5 6))
   '((3 4) (2 4) (2 4) (2 4))))

=>
RTHM-SEQ: num-bars: 4
          num-rhythms: 25
          num-notes: 15
          num-score-notes: 15
          num-rests: 10
          duration: 9.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 1, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: TEST-RS, tag: NIL,
data: ((((3 4) - S S - (E) S (S) (S) S (S) - S E -)
        ((2 4) - S S - (E) - S S (S) S -) ((Q) - S S - (E)) ((E) - S S - (Q))))
```

SYNOPSIS:

```
(defun make-rthm-seq-from-fragments (id fragments references meters
                                     &optional (default-beat 4))
```

28.5.496   rthm-seq/make-rthm-seq-from-unit-multipliers

[rthm-seq] [Functions]

DESCRIPTION:

Given a rhythmic unit, e.g. 32, a list of multipliers (e.g. '(7 9 16)),
and a time signature (e.g. '(4 4)), return a rthm-seq object made up of
bars whose rhythms are multiples of the specified unit by the numbers in
the multipliers list.

At this point the unit should be a whole number divisor of the beat in the
time signature, i.e. quintuple eighths won't work in 4/4.

NB: Setting the auto-beam keyword argument to T can result in errors if
    creating durations longer than 1 beat, as auto-beam will call
    get-beats. :auto-beam is therefore set to NIL by default.

ARGUMENTS:

 - A rhythmic duration unit.
 - A list of multipliers.
 - A time signature.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :tag. A symbol that is another name, description etc. for the given
   object. The tag may be used for identification but not for searching
   purposes. Default = NIL.
 - :auto-beam. T or NIL. When T, the function will attempt to automatically
   set beaming indicators among the resulting rthm-seq-bar objects. This can
   result in errors if the resulting rhythms have a duration of more than 1
   beat. Default = NIL.
 - :id. A symbol that will be the ID of the given object.
   Default = "from-multipliers".
 - :tuplet. An integer or NIL. If an integer, the function will
   automatically place tuplet brackets of that value above beats consisting
   of tuplet rhythms. NB: This function will only place the same value over
   all tuplets. Default = NIL.

RETURN VALUE:

 Returns a rthm-seq object.

EXAMPLE:

```
;; Make a rthm-seq object using the rhythmic unit of a 16th-note, rhythms that
;; are 4, 2, 2, 4 and 4 16th-notes long, and a time signature of 2/4; then
;; print-simple the object returned to see the results.
(let ((rs (make-rthm-seq-from-unit-multipliers 's '(4 2 2 4 4) '(2 4))))
  (print-simple rs))

=>
rthm-seq from-multipliers
(2 4): note Q, note E, note E,
(2 4): note Q, note Q,

;; Make a rthm-seq object using the rhythmic unit of a 32nd note, combinations
;; of irregular duration, and a time signature of 4/4; then print-simple the
;; returned object to see the results.
(let ((rs (make-rthm-seq-from-unit-multipliers 32 '(7 9 16) '(4 4))))
  (print-simple rs))

=>
rthm-seq from-multipliers
(4 4): note E.., note 32, note Q, note H

;; The print-simple output of the above example disregards the ties. We can
;; check to make sure that there are only three attacked rhythms in the result
;; by reading the values of the IS-TIED-FROM and IS-TIED-TO slots, which show
;; that the 32 is tied to the Q
(let ((rs (make-rthm-seq-from-unit-multipliers 32 '(7 9 16) '(4 4))))
  (loop for b in (bars rs)
        collect (loop for r in (rhythms b) collect (is-tied-from r))
        collect (loop for r in (rhythms b) collect (is-tied-to r))))

=> ((NIL T NIL NIL) (NIL NIL T NIL))

;;; Using with a tuplet rhythm ('te) and setting the :tuplet option to 3 so
;;; that triplet brackets are automatically placed.
(let ((rs (make-rthm-seq-from-unit-multipliers 'te '(7 9 16) '(4 4)
                                                 :tuplet 3)))
  (loop for b in (bars rs)
     collect (loop for r in (rhythms b) collect (bracket r))))

=> ((NIL NIL ((1 3)) (1) NIL) (NIL ((1 3)) (1) NIL NIL)
    (NIL NIL ((1 3)) (1) NIL))
```

SYNOPSIS:

```
(defun make-rthm-seq-from-unit-multipliers (unit multipliers time-sig
                                                &key
```

```
                                            ;; a number for brackets over
                                            ;; each beat.
                                            (tuplet nil)
                                            (tag nil)
                                            (auto-beam nil) ; see above
                                            (id "from-multipliers"))
```

28.5.497   rthm-seq/make-rthm-seq-from-unit-multipliers-simp

[rthm-seq] [Functions]

DATE:

 June 6th 2020

DESCRIPTION:

 Make a rthm-seq using lists of multiples of a single unit e.g. '(2 2 1)
 would result in an automatic time signature of 5/8 if passed a unit of 'e
 (or 8). In that case the rhythms would be q q e. Rests may be created by
 passing a number in parentheses.

 See also make-rthm-seq-from-unit-multipliers for a related method.

ARGUMENTS:

 - the ID for the rthm-seq
 - the rhythm unit (symbol, number or rhythm object)
 - the lists of multipliers, each sublist representing a bar

RETURN VALUE:

 a rthm-seq object

EXAMPLE:

```
(print-simple (make-rthm-seq-from-unit-multipliers-simp 1 'e
               '((3 4 1 2) (2 1 2))))
-->
rthm-seq 1
NIL: bar -1: (10 8):
note Q.,
note H,
note E,
note Q,
```

```
NIL: bar -1: (5 8):
note Q,
note E,
note Q,
```

SYNOPSIS:

(defun make-rthm-seq-from-unit-multipliers-simp (id unit multipliers)

28.5.498   rthm-seq/rs-subseq

[rthm-seq] [Methods]

DATE:

 30th September 2013

DESCRIPTION:

 Extract a new rthm-seq object from the bars of an existing rthm-seq.

 NB other -subseq methods are more like Lisp's subseq but as this is for the
 end user it's a little different in the use of its indices.

ARGUMENTS:

 - the original rthm-seq object
 - the start bar (1-based)

OPTIONAL ARGUMENTS:

 - the end bar (1-based and (unlike Lisp's subseq function) inclusive). If
   NIL, we'll use the original end bar. Default = NIL.

RETURN VALUE:

 A new rthm-seq object.

EXAMPLE: SYNOPSIS:

(defmethod rs-subseq ((rs rthm-seq) start &optional end)

28.5.499   rthm-seq/scale

[rthm-seq] [Methods]

DESCRIPTION:

 Scale the durations of the rhythm objects in a given rthm-seq object by the
 specified factor.

 NB: As is evident in the examples below, this method does not replace the
     original data in the rthm-seq object's DATA slot.

ARGUMENTS:

 - A rthm-seq object.
 - A real number that is the scaling factor.

RETURN VALUE:

 Returns a rthm-seq object.

EXAMPLE:

```
;; The method returns a rthm-seq object.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (scale rs 3))

=>
RTHM-SEQ: num-bars: 3
          num-rhythms: 11
          num-notes: 8
          num-score-notes: 9
          num-rests: 2
          duration: 16.5
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
          (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: ((((2 4) Q+E S S) ((E) Q (E)) ((3 8) S S E. S)) PITCH-SEQ-PALETTE
```

```
      ((1 2 3 1 1 2 3 4)))
```

```
;; Create a rthm-seq object, scale the durations by 3 times using the scale
;; method, and print-simple the corresponding slots to see the results
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 1 1 2 3 4))))))
  (print-simple (scale rs 3)))
```

```
=>
rthm-seq NIL
(6 4): note H., note Q., note E., note E.,
(6 4): rest Q., note H., rest Q.,
(9 8): note E., note E., note E., note E.,
```

SYNOPSIS:

```
(defmethod scale ((rs rthm-seq) scaler
                  &optional ignore1 ignore2 ignore3)
```

28.5.500   rthm-seq/set-nth-attack

[rthm-seq] [Methods]

DESCRIPTION:

Sets the value of the nth rhythm object of a given rthm-seq object that
needs an attack; i.e., not a rest and not a tied note.

NB: This method does not check to ensure that the resulting rthm-seq bars
    contain the right number of beats.

ARGUMENTS:

- A zero-based index number for the attacked note to change.
- An event.
- A rthm-seq object.

OPTIONAL ARGUMENTS:

- T or NIL indicating whether to print an error message if the given index
  is greater than the number of attacks (minus 1) in the rthm-seq object
  (default = T).

RETURN VALUE:

 - An event object.

EXAMPLE:

```
;; The method returns an event object
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                             ((e) q (e))
                             ((3 8) s s e. s))
                            :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-attack 2 (make-event 'c4 'q) rs))

=>
EVENT: start-time: NIL, end-time: NIL,
[...]
       pitch-or-chord:
PITCH: frequency: 261.6255569458008, midi-note: 60, midi-channel: NIL
[...]
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: C4, tag: NIL,
data: C4
[...]
       written-pitch-or-chord: NIL
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
[...]
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q

;; Create a rthm-seq object, apply set-nth-attack, print the corresponding
;; slots to see the change
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                             ((e) q (e))
                             ((3 8) s s e. s))
                            :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-attack 2 (make-event 'c4 'q) rs)
  (print-simple rs))

=>
rthm-seq NIL
(2 4): note Q, note E, note S, C4 Q,
(2 4): rest E, note Q, rest E,
(3 8): note S, note S, note E., note S,

;; By default, the method drops into the debugger with an error when the
;; specified index is greater than the number of items in the given rthm-seq
;; object.
```

```
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-attack 11 (make-event 'c4 'q) rs))

=>
rthm-seq::set-nth-attack: Can't set attack 11 as only 8 notes in the rthm-seq
   [Condition of type SIMPLE-ERROR]

;; This error can be suppressed, simply returning NIL, by setting the optional
;; argument to NIL.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                          :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-attack 11 (make-event 'c4 'q) rs nil))

=> NIL
```

SYNOPSIS:

```
(defmethod set-nth-attack (index (e event) (rs rthm-seq)
                           &optional (error t))
```

28.5.501   rthm-seq/set-nth-bar

[rthm-seq] [Methods]

DESCRIPTION:

 Change the contents of the nth rthm-seq-bar object in the given rthm-seq.

ARGUMENTS:

 - A zero-based index number for the bar to change.
 - A rthm-seq-bar object containing the new bar.
 - A rthm-seq object.

RETURN VALUE:

 A rthm-seq-bar object.

EXAMPLE:

```
;; The method returns what is passed to it as the new-bar argument (generally a
;; rthm-seq-bar object.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-bar 1 (make-rthm-seq-bar '((2 4) (s) e (s) q)) rs))

=>
RTHM-SEQ-BAR: time-sig: 0 (2 4), time-sig-given: T, bar-num: -1,
[...]
data: ((2 4) (S) E (S) Q)

;; Create a rthm-seq object, change the second bar (index 1) using the
;; set-nth-bar method, and print the contents of the rhythms data to see the
;; changes.
(let ((rs (make-rthm-seq '((((2 4) q+e s s)
                            ((e) q (e))
                            ((3 8) s s e. s))
                           :pitch-seq-palette ((1 2 3 4 1 1 2 3))))))
  (set-nth-bar 1 (make-rthm-seq-bar '((2 4) (s) e (s) q)) rs)
  (print-simple rs))

=>
rthm-seq NIL
(2 4): note Q, note E, note S, note S,
(2 4): rest S, note E, rest S, note Q,
(3 8): note S, note S, note E., note S,
```

SYNOPSIS:

```
(defmethod set-nth-bar (index new-bar (rs rthm-seq))
```

28.5.502   rthm-seq/split

[rthm-seq] [Methods]

DATE:

 27 Jan 2011

DESCRIPTION:

 Splits the rthm-seq-bar objects of a given rthm-seq object into multiple
 smaller rthm-seq-bar objects, creating a new rthm-seq object with a greater

number of bars than the original. This will only work if the given
rthm-seq-bar objects can be split into whole beats; e.g., a 4/4 bar will
not be split into 5/8 + 3/8.

The keyword arguments :min-beats and :max-beats serve as guidelines rather
than strict cut-offs. In some cases, the method may only be able to
effectively split the given rthm-seq-bar by dividing it into segments that
slightly exceed the length stipulated by these arguments (see example
below).

Depending on the min-beats/max-beats arguments stipulated by the user or
the rhythmic structure of the given rthm-seq-bar objects, the given
rthm-seq-bar or rthm-seq objects may not be splittable, in which case NIL
is returned. If the keyword argument :warn is set to T, a warning will be
also printed in such cases.

NB: This method sets the values of the individual slots but leaves the DATA
slot untouched (for cases in which the user might want to see where the new
data originated from, or otherwise use the old data somehow, such as in a
new rthm-seq object).

ARGUMENTS:

 - A rthm-seq object.

OPTIONAL ARGUMENTS:

keyword arguments
 - :min-beats. This argument takes an integer value to indicate the minimum
   number of beats in any of the new rthm-seq-bar objects created. This
   serves as a guideline only and may occasionally be exceeded in value by
   the method. Default value = 2.
 - :max-beats. This argument takes an integer value to indicate the maximum
   number of beats in any of the new rthm-seq-bar objects created. This
   serves as a guideline only and may occasionally be exceeded in value by
   the method. Default value = 5.
 - :warn. Indicates whether to print a warning if the rthm-seq-bar object is
   unsplittable. Value T = print a warning. Defaults to NIL.

RETURN VALUE:

 A rthm-seq object.

EXAMPLE:

```
;; The method returns a new rthm-seq object
(let ((rs (make-rthm-seq '((((4 4) q e s s (e) e e (e))
                            ((3 4) s s e s e s e. s)
                            ((5 4) h q. e e s s))
                           :pitch-seq-palette ((1 2 3 4 5 6 1 2 3 4 5 6 7 8 1 2
                                                  3 4 5 6))))))
  (split rs))

=>

RTHM-SEQ: num-bars: 5
          num-rhythms: 22
          num-notes: 20
          num-score-notes: 20
          num-rests: 2
          duration: 12.0
          psp-inversions: NIL
          marks: NIL
          time-sigs-tag: NIL
          handled-first-note-tie: NIL
         (for brevity's sake, slots pitch-seq-palette and bars are not printed)
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: ((((4 4) Q E S S (E) E E (E)) ((3 4) S S E S E S E. S)
       ((5 4) H Q. E E S S))
       PITCH-SEQ-PALETTE ((1 2 3 4 5 6 1 2 3 4 5 6 7 8 1 2 3 4 5 6)))

;; Without setting the :min-beats and :max-beats arguments, the following
;; rthm-seq object is broken down from 3 to 5 rthm-seq-bar objects
(let* ((rs (make-rthm-seq '((((4 4) q e s s (e) e e (e))
                             ((3 4) s s e s e s e. s)
                             ((5 4) h q. e e s s))
                            :pitch-seq-palette ((1 2 3 4 5 6 1 2 3 4 5 6 7 8 1 2
                                                   3 4 5 6)))))
       (rssplt (split rs)))
  (print-simple rssplt))

=>
rthm-seq NIL
(2 4): note Q, note E, note S, note S,
(2 4): rest E, note E, note E, rest E,
(3 4): note S, note S, note E, note S, note E, note S, note E., note S,
(2 4): note H,
(3 4): note Q., note E, note E, note S, note S,
```

```
;; Setting :min-beats to 4 affects the resulting subdivisions to larger bars
(let* ((rs (make-rthm-seq '((((4 4) q e s s (e) e e (e))
                             ((3 4) s s e s e s e. s)
                             ((5 4) h q. e e s s))
                            :pitch-seq-palette ((1 2 3 4 5 6 1 2 3 4 5 6 7 8 1 2
                                                      3 4 5 6)))))
       (rssplt (split rs :min-beats 4)))
  (print-simple rssplt))

=>
rthm-seq NIL
(4 4): note Q, note E, note S, note S, rest E, note E, note E, rest E,
(3 4): note S, note S, note E, note S, note E, note S, note E., note S,
(5 4): note H, note Q., note E, note E, note S, note S,

;; Even though :max-beats is set to 2, an occasional 3/4 bar is constructed
(let* ((rs (make-rthm-seq '((((4 4) q e s s (e) e e (e))
                             ((3 4) s s e s e s e. s)
                             ((5 4) h q. e e s s))
                            :pitch-seq-palette ((1 2 3 4 5 6 1 2 3 4 5 6 7 8 1 2
                                                      3 4 5 6)))))
       (rssplt (split rs :max-beats 2)))
  (print-simple rssplt))

=>
rthm-seq NIL
(2 4): note Q, note E, note S, note S,
(2 4): rest E, note E, note E, rest E,
(3 4): note S, note S, note E, note S, note E, note S, note E., note S,
(2 4): note H,
(3 4): note Q., note E, note E, note S, note S,
```

SYNOPSIS:

```
(defmethod split ((rs rthm-seq)
                  &key (min-beats 2) (max-beats 5) warn (clone t))
```

28.5.503   rthm-seq/split-into-single-bars

[rthm-seq] [Methods]

DESCRIPTION:

 Split a rthm-seq into single bar rthm-seqs. The pitch-seq-palette will be
 used to set the pitch-seqs of the new rthm-seqs.

ARGUMENTS:

- a rthm-seq object

OPTIONAL ARGUMENTS:

- whether to clone each bar or just the original. Default = T = clone.

RETURN VALUE:

a list of rthm-seq objects

SYNOPSIS:

```
(defmethod split-into-single-bars ((rs rthm-seq) &optional (clone t))
```

28.5.504   sclist/rthm-seq-bar

[sclist] [Classes]

NAME:

rthm-seq-bar

```
 File:            rthm-seq-bar.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist ->
                  rthm-seq-bar

 Version:         1.0.11

 Project:         slippery chicken (algorithmic composition)

 Purpose:         Implementation of the rthm-seq-bar class, objects of
                  which make up the individual bars that reside in a
                  rhythmic sequence.  This class is responsible for parsing
                  lists containing rhythms and time signatures (but not
                  parsing these things themselves--that is done by separate
                  classes).

 Author:          Michael Edwards: m@michael-edwards.org

 Creation date:   13th February 2001

 $$ Last modified: 13:52:41 Wed Oct  7 2020 CEST

 SVN ID: $Id$
```

28.5.505   rthm-seq-bar/add-half-beat-rest

[rthm-seq-bar] [Methods]

DATE:

 August 25th 2016, Edinburgh

DESCRIPTION:

 Adds a half beat rest at the end of the bar. This is a destructive
 method. It will change the time signature, so a 2/4 bar becomes a 5/8
 bar, with an 1/8th rest at the end.

ARGUMENTS:

 - rthm-seq-bar object

OPTIONAL ARGUMENTS:

 - ignore: don't pass this; it's only there so we can have a bar number
 passed in the slippery-chicken class.

RETURN VALUE:

 The modified rthm-seq-bar object passed as an argument

SYNOPSIS:

(defmethod add-half-beat-rest ((rsb rthm-seq-bar) &optional ignore)

28.5.506   rthm-seq-bar/all-rests?

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Test whether all rhythms in a rthm-seq-bar object are rests.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 T if all rhythms are rests, otherwise NIL

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) (q) (e) (s) (s)))))
  (all-rests? rsb))
```

```
=> T
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
  (all-rests? rsb))
```

```
=> NIL
```

SYNOPSIS:

```
(defmethod all-rests? ((rsb rthm-seq-bar))
```

28.5.507   rthm-seq-bar/auto-beam

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Automatically add beaming indications to the rhythm objects of the given
 rthm-seq-bar object. This will only set one beam group per beat.

 NB: This method does not modify the data slot of the rthm-seq-bar object
     itself. Instead, it modifies the beam value for the individual rhythms.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - The beat basis for the given rthm-seq-bar. This will affect which notes
   get beamed together. This value can be either numeric (4, 8 16 etc.) or
   alphabetic (q, e, s etc). If no beat is given, the method defaults this
   value to NIL and takes the beat from the current time signature.
 - Check-dur. This can be t, nil, #'warn or #'error, where t is the same as
   #'error. If T, the method will make sure there is a complete beat of
   rhythms for each beat of the bar and issue an error if a full beat's
   worth cannot be returned--this may not mean that your bar is
   malformed. Or if you pass a symbol like 'silent the

   duration will be checked and NIL returned if we can't get an exact beat's
   worth of rthms. Default = T.

RETURN VALUE:

 Returns the rthm-seq-bar-object

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) e e s s s s))))
  (auto-beam rsb))

=> NIL

(let ((rsb (make-rthm-seq-bar '((2 4) e e s s s s))))
  (auto-beam rsb)
  (loop for r in (rhythms rsb) collect (beam r)))

=> (1 0 1 NIL NIL 0)

(let ((rsb (make-rthm-seq-bar '((2 4) e e s s s s))))
  (auto-beam rsb 8)
  (loop for r in (rhythms rsb) collect (beam r)))

=> (NIL NIL 1 0 1 0)

(let ((rsb (make-rthm-seq-bar '((2 4) e e s s s s))))
  (auto-beam rsb 8 t)
  (loop for r in (rhythms rsb) collect (beam r)))

=> (NIL NIL 1 0 1 0)

(let ((rsb (make-rthm-seq-bar '((2 4) e e s s s s))))
  (auto-beam rsb 8 nil)
  (loop for r in (rhythms rsb) collect (beam r)))

=> (NIL NIL 1 0 1 0)
```

SYNOPSIS:

```
(defmethod auto-beam ((rsb rthm-seq-bar) &optional (beat nil) (check-dur t))
```

28.5.508   rthm-seq-bar/auto-put-tuplet-bracket-on-beats

[rthm-seq-bar] [Methods]

DESCRIPTION:

Given a rthm-seq-bar object with tuplet rhythms and an indication of which
tuplet value to place, this method will automatically add the appropriate
tuplet bracket to the beats of the bar in the printed score output. If
the TUPLET argument is set to NIL, the method will proceed on the basis of
best-guessing rules.

NB: This method may produce results that encapsulate an entire beat when
    applying brackets to a portion of that beat. Thus bracketing the rhythm
    (e ts ts ts) will return
    { 3 e. ts ts ts } rather than
    (e { 3 ts ts ts })

ARGUMENTS:

 - A rthm-seq-bar object
 - An integer indicating the tuplet value (e.g. 3 for triplets, 5 for
   quintuplets etc.)

RETURN VALUE:

 Returns T.

OPTIONAL ARGUMENTS:

 - An integer indicating beat basis for the bar, or NIL. If NIL (default),
   the beat is taken from the time signature.
 - An integer indicating the beat number within the bar to look for
   tuplets, or T. If T (default), all beats in the bar will be examined for
   possible tuplets.
 - T or NIL to indicate whether to delete the tuplet bracket indicators
   already present in the given rthm-seq-bar object. T = delete.
   Default = T.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) te te te q))))
  (tuplets rsb))

=> NIL

(let ((rsb (make-rthm-seq-bar '((2 4) te te te q))))
  (loop for r in (rhythms rsb) collect (bracket r))
```

```
=> (NIL NIL NIL NIL)

(let ((rsb (make-rthm-seq-bar '((2 4) te te te q))))
  (auto-put-tuplet-bracket-on-beats rsb 3))

=> T

(let ((rsb (make-rthm-seq-bar '((2 4) te te te q))))
  (auto-put-tuplet-bracket-on-beats rsb 3)
  (print (tuplets rsb))
  (print (loop for r in (rhythms rsb) collect (bracket r))))

=>
((3 0 2))
(((1 3)) (-1) (1) NIL)

(let ((rsb (make-rthm-seq-bar '((2 4) te te te q))))
  (auto-put-tuplet-bracket-on-beats rsb nil)
  (tuplets rsb))

=> ((3 0 2))

;;; The method may bracket the entire beat, returning ((3 1 4)) rather than
;;; ((3 2 4))
(let ((rsb (make-rthm-seq-bar '((2 4) q e ts ts ts))))
  (auto-put-tuplet-bracket-on-beats rsb 3)
  (tuplets rsb))

=> ((3 1 4))
```

SYNOPSIS:

```
(defmethod auto-put-tuplet-bracket-on-beats ((rsb rthm-seq-bar) tuplet
                                            &optional
                                            (beat nil)
                                            ;; can be a beat number or t for
                                            ;; all
                                            (beat-number t)
                                            ;; delete the tuplets already
                                            ;; there?
                                            (delete t))
```

28.5.509   rthm-seq-bar/auto-tuplets

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Automatically place the data necessary for tuplet brackets in rthm-seq-bar
 objects that contain tuplet rhythms.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - A function to be performed on fail. Default = #'error.

RETURN VALUE:

 Returns T if successful, and the rsb as a second value in any case

EXAMPLE:

```
;;; Make a rthm-seq-bar object and print the values of the BRACKET slots for
;;; the rhythm objects it contains. Then apply auto-brackets and print the same
;;; again to see the change.
(let ((rsb (make-rthm-seq-bar '((4 4) tq tq tq +q fs fs fs fs fs))))
  (print (loop for r in (rhythms rsb) collect (bracket r)))
  (auto-tuplets rsb)
  (print (loop for r in (rhythms rsb) collect (bracket r))))

=>
(NIL NIL NIL NIL NIL NIL NIL NIL NIL)
(((1 3)) (-1) (1) NIL ((2 5)) (-2) (-2) (-2) (2))
```

SYNOPSIS:

```
(defmethod auto-tuplets ((rsb rthm-seq-bar) &optional (on-fail #'error))
```

28.5.510   rthm-seq-bar/check-beams

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Check the BEAM slots of the event objects within a specified rthm-seq-bar
 object to ensure that every beginning beam indication (slot value of 1) is
 coupled with a corresponding closing beam indication (slot value of 0), and
 print a warning and return NIL if this is not the case.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

```
keyword arguments:
```
 - :auto-beam. T or NIL to indicate the method should apply the auto-beam
   algorithm to the given bar after the check has determined that the
   beaming is wrong (and :on-fail is not NIL). T = auto-beam. Default = NIL.
 - :print. T or NIL to indicate whether the method should print feedback of
   the checking process to the Lisp listener. T = print feedback. Default =
   NIL.
 - :fix. T or NIL to indicate that when a beam has been placed over a rhythm
    with no flags (e.g. a 1/4 note), then we delete beams over that note and
    try again. Default = T.
 - :on-fail. The function that should be applied when the check does not
   pass. May be NIL for no warning/error or #'error if processing should
   stop. Default = #'warn.

RETURN VALUE:

 T if the check passes, otherwise NIL.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                  (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                  (3 ((sax ((3 alto-sax) (4 tenor-sax)))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                    (2 (1 1 1 1 1))
                    (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h e (s) (s) e+s+s))
                                 :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                         (2 ((sax (1 1 1 1 1))))
                         (3 ((sax (1 1 1 1 1)))))))))
  (check-beams (get-bar mini 1 'sax)))

=> T
```

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax)))))
                                 (3 ((sax ((3 alto-sax) (4 tenor-sax)))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h e (s) (s) e+s+s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))))
                        (3 ((sax (1 1 1 1 1)))))))))
  (setf (beam (nth 1 (rhythms (get-bar mini 1 'sax)))) 1)
  (check-beams (get-bar mini 1 'sax)))

=> NIL
```

SYNOPSIS:

```
(defmethod check-beams ((rsb rthm-seq-bar) &key auto-beam print (fix t)
                                                (on-fail #'warn))
```

28.5.511 rthm-seq-bar/check-tuplets

[rthm-seq-bar] [Methods]

DESCRIPTION:

Check the qualities of the tuplets brackets in a given rthm-seq-bar
object to make sure they are all formatted properly (i.e. each starting
tuplet bracket has a closing tuplet bracket etc.). If an error is found,
the method will try to fix it, then re-check, and only issue an error then
if another is found.

NB this won't check whether tuplet brackets start and stop on notes
spanning a whole beat (as that wouldn't make sense with nested tuplets) so
it won't guarantee that a bar can be notated without errors.

ARGUMENTS:

- A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - The function to use if something is not ok with the tuplets. This
   defaults to #'error, but could also be #'warn for example

RETURN VALUE:

 T if all tuplets brackets are ok, otherwise performs the on-fail function
 and returns NIL.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((4 4) { 3 te te te } q q q))))
  (setf (bracket (get-nth-event 2 rsb)) nil)
  (check-tuplets rsb #'warn))

=> rthm-seq-bar::check-tuplets: got a nil bracket when brackets still open.
```

SYNOPSIS:

```
(defmethod check-tuplets ((rsb rthm-seq-bar) &optional (on-fail #'error))
```

28.5.512   rthm-seq-bar/chop

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Creates a list of new rthm-seq-bar objects, with new time signatures, which
 are formed by systematically chopping the bar represented by the current
 rthm-seq-bar into segments.

 The method creates these segments based on chop-point pairs specified in
 the <chop-points> argument, which is a list of 2-element lists, each of
 which specifies the start and end points of a rhythmic span within the
 bounds of a given beat, measured in the unit specified by the <unit>
 argument.

 The chop points specified are used to individually process each beat in the
 given rthm-seq-bar object; thus, chop-points specified for the subdivisions
 of a quarter-note will not work if applied to a 5/8 bar.

 The method fills each newly created rthm-seq-bar object with one rhythmic
 duration that is equal to the length of the bar. If the beginning of the
 given chop segment coincides with an attack in the original bar, the result

is a sounding note; if not, the result is a rest. NB: In this abstraction
of the class for the sake of this documentation, sounding notes will appear
as NIL.

The chop method is the basis for slippery-chicken's feature of
intra-phrasal looping.

NB: The <unit> argument must be a duplet rhythmic value (i.e. 32, 's, 'e
    etc.) and cannot be a tuplet value (i.e. 'te 'fe etc.).

NB: In order for the resulting chopped rhythms to be parsable by LilyPond
    and CMN, there can be no tuplets (triplets etc.) among the rhythms to
    be chopped. Such rhythms will result in LilyPond and CMN errors. This
    has only minimal bearing on any MIDI files produced, however, and these
    can potentially be imported into notation software.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - <chop-points> A list of integer pairs, each of which delineates a segment
   of the beat of the given rthm-seq-bar object measured in the rhythmic
   unit specified by the <unit> argument. Thus, if all possible spans of
   sixteenth-notes within a quarter-note, starting from the first sixteenth,
   were delineated, they would span from 1 to 4 (the full quarter), 1 to 3
   (the first dotted 8th of the quarter), 1 to 2 (the first 8th) and 1 to 1
   (the first 16th of the quarter); the process could continue then with all
   rhythmic durations contained within the bounds of the same quarter
   starting on the second 16th, etc. The default chop-points for a quarter
   are '((1 4) (1 3) (1 2) (2 4) (2 3) (3 4) (1 1) (2 2) (3 3) (4 4)).
 - <unit>. The rhythmic duration that serves as the unit of measurement for
   the chop points. Default = 's.
 - <rthm-seq-id>. A symbol that will be the ID for the list created.

RETURN VALUE:

 A list of rthm-seq-bar objects.

EXAMPLE:

;; Systematically subdivide each quarter-note of a 2/4 bar containing two ; ;
;; quarter-notes into all possible segments whose durations are multiples of a ; ;
;; sixteenth-note unit, and print-simple the resulting list. The quarter-note ; ;

```
;; subdivision is re-specified here sightly differently to the default for the ; ;
;; sake of systematic clarity. Only those segments whose start point coincide ; ;
;; with an attack in the original bar, i.e. those that begin on the first ; ;
;; sixteenth of each  beat, will be assigned a NIL (which will later become ; ;
;; a sounding note); all others are assigned a rest. ; ;

(let* ((rsb (make-rthm-seq-bar '((2 4) q q)))
(ch (chop rsb
'((1 4) (1 3) (1 2) (1 1)
(2 4) (2 3) (2 2)
(3 4) (3 3)
(4 4))
's)))
(loop for b in ch do (print-simple b)))

=>
(1 4): NIL Q,
(3 16): NIL E.,
(1 8): NIL E,
(1 16): NIL S,
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
(1 8): rest 8,
(1 16): rest 16,
(1 16): rest 16,
(1 4): NIL Q,
(3 16): NIL E.,
(1 8): NIL E,
(1 16): NIL S,
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
(1 8): rest 8,
(1 16): rest 16,
(1 16): rest 16,

;; The same thing, but returning all possible segments within the bounds of a ; ;
;; quarter-note whose durations that are multiple of an 8th-note unit ; ;
(let* ((rsb (make-rthm-seq-bar '((2 4) q q)))
(choprsb (chop rsb
'((1 2) (1 1) (2 2))
'e)))
(loop for b in choprsb do (print-simple b)))

=>
```

```
(1 4): NIL Q,
(1 8): NIL E,
(1 8): rest 8,
(1 4): NIL Q,
(1 8): NIL E,
(1 8): rest 8,

;; Adapt the 16th-note example above to a starting rthm-seq-bar object with ; ;
;; more complex rhythmic content. Note here, too, that the rthm-seq-bar object ; ;
;; being segmented contains rhythmic durations smaller than the <unit> ; ;
;; argument.                                    ; ;
(let* ((rsb (make-rthm-seq-bar '((4 4) - (s) (32) 32 (s) s - - +s+32 (32) (e) -
(q) (s) s (e))))
(choprsb (chop rsb
'((1 4) (1 3) (1 2) (1 1)
(2 4) (2 3) (2 2)
(3 4) (3 3)
(4 4))
's)))
(loop for b in choprsb do (print-simple b)))

=>
(1 4): rest S, rest 32, NIL 32, rest S, NIL S,
(3 16): rest S, rest 32, NIL 32, rest S,
(1 8): rest S, rest 32, NIL 32,
(1 16): rest 16,
(3 16): rest 32, NIL 32, rest S, NIL S,
(1 8): rest 32, NIL 32, rest S,
(1 16): rest 32, NIL 32,
(1 8): rest S, NIL S,
(1 16): rest 16,
(1 16): NIL S,
(1 4): rest 4,
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
(1 8): rest 8,
(1 16): rest 16,
(1 16): rest 16,
(1 4): rest 4,
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
```

```
(3 16): rest 16/3,
(1 8): rest 8,
(1 16): rest 16,
(1 8): rest 8,
(1 16): rest 16,
(1 16): rest 16,
(1 4): rest S, NIL S, rest E,
(3 16): rest S, NIL S, rest S,
(1 8): rest S, NIL S,
(1 16): rest 16,
(3 16): NIL S, rest E,
(1 8): NIL S, rest S,
(1 16): NIL S,
(1 8): rest 8,
(1 16): rest 16,
(1 16): rest 16,

;; The same again with a <unit> of eighths ; ;
(let* ((rsb (make-rthm-seq-bar '((4 4) - (s) (32) 32 (s) s - - +s+32 (32) (e) -
(q) (s) s (e))))
(choprsb (chop rsb
'((1 2) (1 1) (2 2))
'e)))
(loop for b in choprsb do (print-simple b)))

=>
(1 4): rest S, rest 32, NIL 32, rest S, NIL S,
(1 8): rest S, rest 32, NIL 32,
(1 8): rest S, NIL S,
(1 4): rest 4,
(1 8): rest 8,
(1 8): rest 8,
(1 4): rest 4,
(1 8): rest 8,
(1 8): rest 8,
(1 4): rest S, NIL S, rest E,
(1 8): rest S, NIL S,
(1 8): rest 8,
```

SYNOPSIS:

```
(defmethod chop ((rsb rthm-seq-bar)
                 &optional chop-points (unit 's) rthm-seq-id)
```

28.5.513   rthm-seq-bar/consolidate-notes

[rthm-seq-bar] [Methods]

DESCRIPTION:

Combine consecutive tied notes into one (or a few) notes of a longer
rhythmic duration.

NB: This method is the core method that is called for rthm-seq objects or
    slippery-chicken objects, at which point it takes ties (and perhaps
    another couple of things) into consideration, after the tie slots
    etc. have been updated. As such, though it will
    work to a certain degree when called directly on a rthm-seq-bar object,
    it should primarily be used when getting a rthm-seq-bar from within a
    rthm-seq object or slippery-chicken object.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the method should make sure that an exact
   beat's worth of rhythms is handled. T = check durations. Default = NIL.

RETURN VALUE:

 The rthm-seq-bar object.

EXAMPLE:

```
;;; Create a slippery-chicken object, print-simple a bar from that object,
;;; apply the consolidate-notes method to that bar, and print-simple that bar
;;; again to see the changes.

(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((gs4 af4 bf4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e +e +e +e e +s +s +s e.))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1)))))))))
```

```
  (print-simple (get-bar mini 2 'vn))
  (consolidate-notes (get-bar mini 2 'vn))
  (print-simple (get-bar mini 2 'vn)))

=>
(4 4): GS4 E+, +GS4 E+, +GS4 E+, +GS4 E, AF4 E+, +AF4 S+, +AF4 S+, +AF4 S,
BF4 E.,
(4 4): GS4 H, AF4 Q+, +AF4 S, BF4 E.,
```

SYNOPSIS:

```
(defmethod consolidate-notes ((rsb rthm-seq-bar)
                                 ;; MDE Wed Nov 28 11:40:59 2012 -- added auto-beam
                                 &optional check-dur beat (auto-beam t))
```

28.5.514   rthm-seq-bar/consolidate-rests

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Consolidate two or more consecutive rests into one longer rhythmic
 unit. This method works on the basis of beats, striving to consolidate into
 beats first.

 NB: The user may find it helpful to adjust the :beat and :min values, and
     even to call the method more than once consecutively. For multiple
     calls, the method consolidate-rests-max may also be useful.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 keyword arguments
 - :beat. The beat basis into which rests are to be consolidated. If no
   value is given for this option, the method will take the beat from the
   time signature.
 - :min. A seldom-used argument that will only make a difference when there
   are a number of rests of the same duration followed by a note. This is
   then the minimum duration that such rests may have if they are to be
   consolidated. Default = NIL.
 - :warn. T or NIL to indicate whether the method should print a warning to
   the Lisp listener if it is mathematically unable to consolidate the

   rests. T = print warning. Default = NIL.
 - :auto-tuplets. Whether to force a call to the auto-tuplets method before
   returning. This will happen by default anyway if check-tuplets fails, but
   sometimes that doesn't fail, exactly, but auto-tuplets still makes
   sense. If, for example, Lilypond fails with strange error messages,
   setting this to T might help. Default = NIL.

RETURN VALUE:

 The rthm-seq-bar object

EXAMPLE:

```
;;; Returns a list of rhythm/event objects
(let ((rsb (make-rthm-seq-bar '((4 4) (e) (e) (e) (e) (e) (s) (s) (s) e.))))
  (consolidate-rests rsb))

=>
(
EVENT: start-time: NIL, end-time: NIL,
[...]
data: 4
[...]
EVENT: start-time: NIL, end-time: NIL,
[...]
data: 4
[...]
EVENT: start-time: NIL, end-time: NIL,
[...]
data: 4
[...]
RHYTHM: value: 16.000, duration: 0.250, rq: 1/4, is-rest: T,
[...]
data: S
[...]
RHYTHM: value: 5.333, duration: 0.750, rq: 3/4, is-rest: NIL,
[...]
data: E.
)

;;; Consolidating on the basis of the time-signature's beat by default
(let ((rsb (make-rthm-seq-bar '((4 4) (e) (e) (e) (e) (e) (s) (s) (s) e.))))
  (consolidate-rests rsb)
  (loop for r in (rhythms rsb) collect (data r)))

=> (4 4 4 S E.)
```

```
;; Changing the :beat may effect the outcome
(let ((rsb (make-rthm-seq-bar '((4 4) (e) (e) (e) (e) (e) (s) (s) (s) e.)))) 
  (consolidate-rests rsb :beat 2)
  (loop for r in (rhythms rsb) collect (data r)))

=> (2 E E S E.)

;; Calling multiple times may further consolidate the results
(let ((rsb (make-rthm-seq-bar '((2 2) (e) (e) (e) (e) (e) (s) (s) (s) e.))))
  (consolidate-rests rsb)
  (print (loop for r in (rhythms rsb) collect (data r)))
  (consolidate-rests rsb)
  (print (loop for r in (rhythms rsb) collect (data r))))

=>
(2 E E S E.)
(2 Q S E.)
```

SYNOPSIS:

```
(defmethod consolidate-rests ((rsb rthm-seq-bar)
                              &key beat min warn auto-tuplets)
```

28.5.515   rthm-seq-bar/consolidate-rests-max

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Similar to consolidate-rests, but calls that method repeatedly until no
 more changes can be made to the given rthm-seq-bar object.

 NB This will still only reduce rests down to a maximum of a beat. If you
    need e.g. two quarter rests reduced to a single half rest in a 4/4 bar,
    specify :beat 2

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :beat. The beat basis into which rests are to be consolidated. If no

value is given for this option, the method will take the beat from the
time signature.
- :min. A seldom-used argument that will only make a difference when there
  are a number of rests of the same duration followed by a note. This is
  then the minimum duration that such rests may have if they are to be
  consolidated. Default = NIL.
- :warn. T or NIL to indicate whether the method should print a warning to
  the Lisp listener if it is mathematically unable to consolidate the
  rests. T = print warning. Default = NIL.

RETURN VALUE:

 The rthm-seq-bar-object

EXAMPLE:

```
;;; Two examples with the same result; the first calling consolidate-rests
;;; twice, the second calling consolidate-rests-max
(let ((rsb (make-rthm-seq-bar '((2 2) (e) (e) (e) (e) (e) (s) (s) (s) e.))))
  (consolidate-rests rsb)
  (consolidate-rests rsb)
  (loop for r in (rhythms rsb) collect (data r)))

=> (2 Q S E.)

(let ((rsb (make-rthm-seq-bar '((2 2) (e) (e) (e) (e) (e) (s) (s) (s) e.))))
  (consolidate-rests-max rsb)
  (loop for r in (rhythms rsb) collect (data r)))

=> (2 Q S E.)
```

SYNOPSIS:

```
(defmethod consolidate-rests-max ((rsb rthm-seq-bar) &key beat min warn)
```

28.5.516   rthm-seq-bar/delete-beams

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Remove any beaming indications from the rthm-seq-bar object.

 NB: This method changes the data for the rthm-seq-bar object's BEAMS slot
     and the individual BEAM slots of the RHYTHMs contained within the

rthm-seq-bar's RHYTHMS slot. It does not change the value of the
rthm-seq-bar's DATA slot.

NB: Neither the presence nor absence of beams are not reflected in the
output of the print-simple method.

ARGUMENTS:

- A rthm-seq-bar object.

RETURN VALUE:

Returns T.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) - s s - s - s s s - s s))))
  (delete-beams rsb))

=> T

(let ((rbs (make-rthm-seq-bar '((2 4) - s s - s - s s s - s s))))
  (delete-beams rsb)
  (beams rsb))

=> NIL

(let ((rsb (make-rthm-seq-bar '((2 4) - s s - s - s s s - s s))))
  (delete-beams rsb)
  (loop for r in (rhythms rbs) collect (beam r)))

=> (NIL NIL NIL NIL NIL NIL NIL NIL)

(let ((rbs (make-rthm-seq-bar '((2 4) - s s - s - s s s - s s))))
  (delete-beams rsb)
  (print rsb))

=>
RTHM-SEQ-BAR: time-sig: 1 (2 4)
              time-sig-given: T
              bar-num: -1
              old-bar-nums: NIL
              write-bar-num: NIL
              start-time: -1.0
              start-time-qtrs: -1.0
```

```
                    is-rest-bar: NIL
                    multi-bar-rest: NIL
                    show-rest: T
                    notes-needed: 8
                    tuplets: NIL
                    nudge-factor: 0.35
                    beams: NIL
[...]
NAMED-OBJECT: id: NIL, tag: NIL,
data: ((2 4) - S S - S - S S S - S S)
```

SYNOPSIS:

```
(defmethod delete-beams ((rsb rthm-seq-bar))
```

28.5.517   rthm-seq-bar/delete-marks

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Delete all marks from the rhythm (or event) objects contained within a given
 rthm-seq-bar object.

 ARGUMENT
 - A rthm-seq-bar object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Create a rthm-seq-bar object and print the contents of the MARKS slots of
;; the contained event objects to see they're set to NIL by default. Fill them
;; each with a 's (staccato) mark and print the results. Apply the delete-marks
;; method and print the results again to see that the values have been reset to
;; NIL.
(let ((rsb (make-rthm-seq-bar
             (list
              '(3 8)
              (make-event 'cs4 'e)
              (make-event 'cs4 'e)
              (make-event 'cs4 'e)))))
  (print (loop for e in (rhythms rsb) collect (marks e)))
```

```
  (loop for e in (rhythms rsb) do (add-mark-once e 's))
  (print (loop for e in (rhythms rsb) collect (marks e)))
  (delete-marks rsb)
  (print (loop for e in (rhythms rsb) collect (marks e))))
```

```
=>
(NIL NIL NIL)
((S) (S) (S))
(NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod delete-marks ((rsb rthm-seq-bar))
```

28.5.518   rthm-seq-bar/delete-tuplets

[rthm-seq-bar] [Methods]

DESCRIPTION:

Removes all indications for tuplet brackets from a given rthm-seq-bar
object.

NB: This method does not alter the tuplet rhythmic durations; it only
removes the tuplet bracket from the score.

ARGUMENTS:

 - A rthm-seq-bar.

RETURN VALUE:

 The rthm-seq-bar object

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) { 3 te te te } q))))
  (tuplets rsb))
```

```
=> ((3 0 2))
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) { 3 te te te } q))))
  (delete-tuplets rsb))
```

```
=> NIL
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) { 3 te te te } q))))
  (delete-tuplets rsb)
  (tuplets rsb))
```

```
=> NIL
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) { 3 te te te } q))))
  (loop for r in (rhythms rsb) collect (bracket r)))
```

```
=> (((1 3)) (-1) (1) NIL)
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) { 3 te te te } q))))
  (delete-tuplets rsb)
  (loop for r in (rhythms rsb) collect (bracket r)))
```

```
=> (NIL NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod delete-tuplets ((rsb rthm-seq-bar))
```

28.5.519   rthm-seq-bar/delete-written

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Delete the contents of the WRITTEN-PITCH-OR-CHORD slot of a pitch object
 within a given event object and reset to NIL.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Create a rthm-seq-bar object consisting of events and print the contents of ; ; ; ; ; ;
;; the WRITTEN-PITCH-OR-CHORD slots to see they're set to NIL. Apply the ; ; ; ; ; ;
;; set-written method with a value of -2 and print the contents of the ; ; ; ; ; ;
;; WRITTEN-PITCH-OR-CHORD slots to see the data of the newly created pitch ; ; ; ; ; ;
```

```
;; objects. Apply the delete-written method and print the contents of the ; ; ; ; ; ; ;
;; WRITTEN-PITCH-OR-CHORD slots to see they're empty. ; ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(print (loop for p in (rhythms rsb)
collect (written-pitch-or-chord p)))
(set-written rsb -2)
(print (loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))
(delete-written rsb)
(print (loop for p in (rhythms rsb)
collect (written-pitch-or-chord p))))

  =>
  (NIL NIL NIL)
  (B3 B3 B3)
  (NIL NIL NIL)
```

SYNOPSIS:

```
(defmethod delete-written ((rsb rthm-seq-bar))
```

28.5.520   rthm-seq-bar/enharmonic

[rthm-seq-bar] [Methods]

DESCRIPTION:

Change the pitches of the events within a given rthm-seq-bar object to
their enharmonic equivalents.

In its default form, this method only applies to note names that already
contain an indication for an accidental (such as DF4 or BS3), while
"white-key" note names (such as B3 or C4) will not produce an enharmonic
equivalent. In order to change white-key pitches to their enharmonic
equivalents, set the :force-naturals argument to T.

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :written. T or NIL to indicate whether the test is to handle the
  written or sounding pitch in the event. T = written.
  Default = NIL.
- :force-naturals. T or NIL to indicate whether to force "natural"
  note names that contain no F or S in their name to convert to
  their enharmonic equivalent (e.g. B3 = CF4). Default = NIL.
- :pitches. All sharp/flat pitches are changed by default but if a
  list of pitch objects or symbols is given, then only those
  pitches will be changed.  Note that if written is T, then this
  pitch list should be the written not sounding pitches.
  Default = NIL.
```

RETURN VALUE:

```
 Always returns T.
```

EXAMPLE:

```
;; The method returns T.                  ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list '(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(enharmonic rsb))

  => T


;; Create a rthm-seq-bar object with events, apply the enharmonic method, and ; ; ; ; ;
;; print the corresponding slots to see the changes ; ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(enharmonic rsb)
(loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))

  => (DF4 DF4 DF4)


;; By default, the method will not change white-key pitches ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
```

```
'(3 8)
(make-event 'c4 'e)
(make-event 'c4 'e)
(make-event 'c4 'e)))))
(enharmonic rsb)
(loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))

  => (C4 C4 C4)

;; This can be forced by setting the :force-naturals argument to T ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'c4 'e)
(make-event 'c4 'e)
(make-event 'c4 'e)))))
(enharmonic rsb :force-naturals t)
(loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))

  => (BS3 BS3 BS3)

;; Apply the set-written method to fill the WRITTEN-PITCH-OR-CHORD slot, print ; ; ; ; ;
;; its contents, apply the enharmonic method with the :written keyword argument ; ; ; ; ;
;; set to T, then print the pitch data of the same slot again to see the ; ; ; ; ;
;; change.                              ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-written rsb -3)
(print (loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))
(enharmonic rsb :written t)
(print (loop for p in (rhythms rsb)
collect (get-pitch-symbol p))))

  =>
  (BF3 BF3 BF3)
  (AS3 AS3 AS3)
```

SYNOPSIS:

```
(defmethod enharmonic ((rsb rthm-seq-bar) &key written force-naturals
                       ;; MDE Wed Apr 18 11:34:01 2012
                       pitches)
```

28.5.521   rthm-seq-bar/events-update-time

[rthm-seq-bar] [Functions]

DATE:

 March 16th 2016, Edinburgh

DESCRIPTION:

 Update the start-time and related slots of any events for which timing
 information is missing or out-of-sync.

ARGUMENTS:

 - a list of event objects

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start-time. The start time in seconds of the first event. Default = 0.0
 - :start-time-qtrs. The start time in quarter notes of the first event (used
   in MIDI files). If NIL, then this will be calculated as a function of
   :start-time and :tempo. If not NIL, this will mean start-time is ignored
   when writing MIDI files. Default = nil.
 - :tempo. The tempo in beats per minute (or a tempo object)
 - :max-start-time. A maximum time in seconds above which we stop processing
   (and returning) events.

RETURN VALUE:

 two values: the list of events with their times updated, and the time the
 last event finishes.

EXAMPLE:

```
(event-list-to-midi-file
  (events-update-time (make-events2 '(q q q) '(g4 g4 g4) 1))
  :midi-file "/tmp/test.mid" :start-tempo 120)
```

SYNOPSIS:

```
(defun events-update-time (events &key (start-time 0.0) start-time-qtrs
                                     max-start-time (tempo 60.0))
```

28.5.522   rthm-seq-bar/fill-with-rhythms

[rthm-seq-bar] [Methods]

DESCRIPTION:

Any rhythms (or event objects) in the existing rthm-seq-bar object will be
deleted, and then rhythm (or event) objects will be taken one by one from
the <rhythms> argument until the bar is full.

If too few rhythm or event objects are given, a warning will be printed
that there are too few beats in the bar.

If there are too many and the last rhythm or event object to be placed in
the bar fills out the bar evenly, no warning is printed and the remaining
rhythm or event objects are ignored. If :is-full-error is T (default) and
the last rhythm or event object that the method attempts to place in the
bar is too long to fit evenly into the bar, the method will drop into the
debugger with an error. If in this case :is-full-error is NIL the bar will
remain underfull and the index of the last rhythm added before over-filling
will be returned.

The :transposition, :midi-channel, and :microtones-midi-channel arguments
can only be used in conjunction with event objects.

The number of rhythms (or event objects) used is returned.

NB: This method does not change the DATA slot of the rthm-seq-bar object
    itself to reflect the new rhythms. Instead, it changes the contents of
    the RHYTHMS slot within that object and changes the DATA of the
    rthm-seq-bar object to NIL. It also assigns the ID of the named-object
    to "rhythms-inserted-by-fill-with-rhythms".

ARGUMENTS:

 - A rthm-seq-bar object.
 - A list of rhythm objects or event objects.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :transposition. An integer or NIL to indicate the transposition in
   semitones for written pitches of any event objects passed. If NIL, no
   written-pitches will be created. Default = NIL.
 - :midi-channel. An integer that will be used to set the MIDI-CHANNEL slot
   of any event objects passed. Default = 0.

- :microtones-midi-channel. An integer that is the MIDI channel that will
  be assigned to event objects for microtonal MIDI pitches. NB: This value
  is only set when attached to event objects within a slippery-chicken
  object. Default = 0. NB: See player.lsp/make-player for details on
  microtones in MIDI output.
- :new-id. An optional ID for new rhythm or event objects added.
  Default = "rhythms-inserted-by-fill-with-rhythms".
- :warn. T or NIL to indicate whether a warning should be printed if there
  are not enough rhythms to create a full bar. T = warn. Default = T.
- :is-full-error. T or NIL to indicate whether the last rhythm or event
  object that the method attempts to add to the bar is too long to fit
  evenly into the bar. T = drop into the debugger with an error if this is
  the case. Default = T.

RETURN VALUE:

The number of rhythm or event objects used, which could be 0. NIL will be
returned if, for example, an empty list of rhythms were passed, so do check
for a NIL result.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((3 4) q q q))))
  (fill-with-rhythms rsb (loop for r in '(e e e e e e)
                          collect (make-rhythm r))))

=> 6

(let ((rsb (make-rthm-seq-bar '((3 4) q q q))))
  (fill-with-rhythms rsb (loop for r in '(e e e e e e)
                          collect (make-rhythm r)))
  (print-simple rsb))

=> NIL
(3 4): note E, note E, note E, note E, note E, note E,

(let ((rsb (make-rthm-seq-bar '((3 4) q q q))))
  (fill-with-rhythms rsb (loop for r in '(e e e e e e)
                          collect (make-rhythm r)))
  (print rsb))

=>
RTHM-SEQ-BAR: time-sig: 0 (3 4)
              time-sig-given: T
              bar-num: -1
              old-bar-nums: NIL
```

```
                          write-bar-num: NIL
                          start-time: -1.0
                          start-time-qtrs: -1.0
                          is-rest-bar: NIL
                          multi-bar-rest: NIL
                          show-rest: T
                          notes-needed: 6
                          tuplets: NIL
                          nudge-factor: 0.35
                          beams: NIL
                          current-time-sig: 0
                          write-time-sig: T
                          num-rests: 0
                          num-rhythms: 6
                          num-score-notes: 6
                          rhythms: (
RHYTHM: value: 8.0, duration: 0.5, rq: 1/2, is-rest: NIL, score-rthm: 8.0,
        undotted-value: 8, num-flags: 1, num-dots: 0, is-tied-to: NIL,
        is-tied-from: NIL, compound-duration: 0.5, is-grace-note: NIL,
        needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
        rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 8,
        tuplet-scaler: 1, grace-note-duration: 0.05,
LINKED-NAMED-OBJECT: previous: NIL
                     this: NIL
                     next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E
[...]
NAMED-OBJECT: id: "rhythms-inserted-by-fill-with-rhythms", tag: NIL,
data: NIL

;;; Using the :transpositions and :midi-channel arguments
(let ((rsb (make-rthm-seq-bar '((4 4) q q q q))))
  (fill-with-rhythms rsb (loop for r in '(h q e s s)
                               for p in '(c4 dqs4 e4 gqf4 a4)
                               collect (make-event p r))
                     :microtones-midi-channel 12
                     :transposition -14
                     :midi-channel 11)
  (print
   (loop for e in (rhythms rsb)
      collect (data (pitch-or-chord e))))
  (print
   (loop for e in (rhythms rsb)
      collect (data (written-pitch-or-chord e))))
  (print
```

```
   (loop for e in (rhythms rsb)
      collect (midi-channel (pitch-or-chord e)))))

=>
(C4 DQS4 E4 GQF4 A4)
(D5 EQS5 FS5 AQF5 B5)
(11 12 11 12 11)
```

SYNOPSIS:

```
(defmethod fill-with-rhythms ((rsb rthm-seq-bar) rhythms
                              &key
                              ;; 24.3.11 add this too to make sure written
                              ;; pitch is set--this is the instrument
                              ;; transposition e.g. -14 for bass clarinet
                              transposition
                              (midi-channel 0)
                              (microtones-midi-channel 0)
                              (new-id "rhythms-inserted-by-fill-with-rhythms")
                              (warn t)
                              (is-full-error t))
```

28.5.523   rthm-seq-bar/force-all-rests

[rthm-seq-bar] [Methods]

DATE:

 November 7th 2018, Heidhausen

DESCRIPTION:

 Makes all events in the bar rests. This differs from force-rest-bar in that
 it doesn't replace the events with a single full-bar rest

ARGUMENTS:

 - a rthm-seq-bar object

OPTIONAL ARGUMENTS:

 - T or NIL to indicate that any marks (e.g. dynamics) should be retained,
   whether or not these make sense on rests. The exception is that slurs
   won't be retained. Default = NIL.

RETURN VALUE:

 the modified rthm-seq-bar object

SYNOPSIS:

(defmethod force-all-rests ((rsb rthm-seq-bar) &optional retain-marks)

28.5.524   rthm-seq-bar/force-rest-bar

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Force all rhythms of a rthm-seq-bar object to be replaced by rest.

 NB: This method changes the value of the RHYTHMS slot of the rthm-seq-bar
     but not the value of the rthm-seq-bar DATA slot.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 Returns a rthm-seq-bar object.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
  (force-rest-bar rsb))

=>
RTHM-SEQ-BAR: time-sig: 1 (2 4)
              time-sig-given: T
              bar-num: -1
              old-bar-nums: NIL
              write-bar-num: NIL
              start-time: -1.0
              start-time-qtrs: -1.0
              is-rest-bar: T
[...]
RHYTHM: value: 2.0, duration: 2.0, rq: 2, is-rest: T,
[...]
NAMED-OBJECT: id: NIL, tag: NIL,
```

```
data: ((2 4) Q E S S)

(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
  (force-rest-bar rsb)
  (print-simple rsb))

=>
(2 4): rest 2,
```

SYNOPSIS:

```
(defmethod force-rest-bar ((rsb rthm-seq-bar))
```

28.5.525   rthm-seq-bar/get-beats

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Try to organise the events of a bar into sublists of a beat's worth of
 events.

 If beat is nil, we'll get the beat from the time-sig. If check-dur is T
 we'll make sure we get a complete beat of rhythms for each beat of the bar
 Since May 1 2012, check-dur can be T, NIL, #'warn or #'error, where t is
 the same as #'error. If you pass a symbol like 'silent the duration will be
 checked and NIL returned if we can't get an exact beat's worth of rthms.

ARGUMENTS:

 - a rthm-seq-bar object

OPTIONAL ARGUMENTS:

 - the beat, as a symbol or rhythm object (see above)
 - whether to check the duration of the beats to see if they're full (see
 above)

RETURN VALUE:

 A list of lists of event/rhythm objects

EXAMPLE:

NB the elements returned will actually be rhythm objects but I'll just show the
rhythm symbol for clarity:
(get-beats (make-rthm-seq-bar '((4 4) s x 4 q e e s e.)))
-->
((s s s s) (q) (e e) (s e.))

SYNOPSIS:

(defmethod get-beats ((rsb rthm-seq-bar) &optional beat check-dur)

28.5.526   rthm-seq-bar/get-last-attack

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Gets the rhythm object for the last note that needs an attack (i.e. not a
 rest and not a tied note) in a given rthm-seq-bar object.

ARGUMENTS:

 - The given rthm-seq-bar object.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print a warning message if the given index
   (minus one) is greater than the number of attacks in the RHYTHMS list
   (default = T). This is a carry-over argument from the get-nth-attack
   method called within the get-last-attack method and not likely to be
   needed for use with get-last-attack.

RETURN VALUE:

 A rhythm object.

 Returns NIL if the given index is higher than the highest possible index of
 attacks in the given rthm-seq-bar object.
 Get the rhythm object of the last

EXAMPLE:

(let ((rsb (make-rthm-seq-bar '((3 4) q+e (e) s (s) e))))
(get-last-attack rsb))

```
=>
RHYTHM: value: 8.0, duration: 0.5, rq: 1/2, is-rest: NIL, score-rthm: 8.0,
undotted-value: 8, num-flags: 1, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 0.5, is-grace-note: NIL,
needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 8,
tuplet-scaler: 1, grace-note-duration: 0.05,
LINKED-NAMED-OBJECT: previous: NIL
this: NIL
next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E
```

SYNOPSIS:

```
(defmethod get-last-attack ((rsb rthm-seq-bar) &optional (warn t))
```

28.5.527   rthm-seq-bar/get-last-attacks

[rthm-seq-bar] [Methods]

DATE:

 July 24th 2015, Glenferness

DESCRIPTION:

 Return the last nth attacks in a bar, with any notes tied from them.

ARGUMENTS:

 - a rthm-seq-bar object
 - an integer indicating how many attacks to return

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether a warning should be issued if the required
   number of attacks cannot be returned.

RETURN VALUE:

 A list of lists of rhythm or event objects. The length of the outer list
 will be the same as the second argument. The sublists will contain the
 attack event with any subsequent events tied from it. NB if a request is
 made for more attacks than the bar contains the method returns NIL (rather
 than all the attacks in the bar).

SYNOPSIS:

```
(defmethod get-last-attacks ((rsb rthm-seq-bar) how-many &optional (warn t))
```

28.5.528   rthm-seq-bar/get-last-event

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Get the last event object (or rhythm object) of a given rthm-seq-bar
 object.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 Returns a rhythm object.

EXAMPLE:

```
;; Returns a rhythm object.
(let ((rsb (make-rthm-seq-bar '((2 4) s s e q))))
(get-last-event rsb))

=>
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
score-rthm: 4.0f0, undotted-value: 4, num-flags: 0, num-dots: 0,
is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q
```

SYNOPSIS:

```
(defmethod get-last-event ((rsb rthm-seq-bar))
```

28.5.529   rthm-seq-bar/get-nearest-by-start-time

[rthm-seq-bar] [Methods]

DATE:

December 16th 2019, Essen Werden

DESCRIPTION:

 Find the event with the closest start-time to a given target.

ARGUMENTS:

 - a rthm-seq-bar object, with times appropriately initialised (e.g. by
   make-slippery-chicken)
 - the target: either start time in seconds (float) or an event object

RETURN VALUE:

 two values: the closest event and it's position in the bar (0-based)

SYNOPSIS:

(defmethod get-nearest-by-start-time ((rsb rthm-seq-bar) target)

28.5.530   rthm-seq-bar/get-nth-attack

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Gets the rhythm object for the nth note in a given rthm-seq-bar that needs
 an attack, i.e. not a rest and not tied.

ARGUMENTS:

 - The zero-based index number indicating which attack is sought.
 - The given rthm-seq-bar object in which to search.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print a warning message if the given index
   is greater than the number of attacks in the RHYTHMS list (minus one to
   compensate for the zero-based indexing) (default = T).

RETURN VALUE:

 A rhythm object.

 Returns NIL if the given index is higher than the highest possible index of
 attacks in the given rthm-seq-bar object.

EXAMPLE:

```
;; The method returns a rhythm object when successful
(let ((rsb (make-rthm-seq-bar '((3 4) q+e (e) s (s) e))))
(get-nth-attack 0 rsb))

=>
RHYTHM: value: 4.0, duration: 1.0, rq: 1, is-rest: NIL, score-rthm: 4.0,
undotted-value: 4, num-flags: 0, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 1.0, is-grace-note: NIL,
needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 4,
tuplet-scaler: 1, grace-note-duration: 0.05,
LINKED-NAMED-OBJECT: previous: NIL
this: NIL
next: NIL
NAMED-OBJECT: id: "Q", tag: NIL,
data: Q

(let ((rsb (make-rthm-seq-bar '((3 4) q+e (e) s (s) e))))
(data (get-nth-attack 1 rsb)))

=> S

(Let ((rsb (make-rthm-seq-bar '((3 4) q+e (e) s (s) e))))
(get-nth-attack 3 rsb))

=> NIL
WARNING: rthm-seq-bar::get-nth-attack:  index (3) < 0 or >= notes-needed (3)

(Let ((rsb (make-rthm-seq-bar '((3 4) q+e (e) s (s) e))))
(get-nth-attack 3 rsb nil))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-attack (index (rsb rthm-seq-bar) &optional (warn t))
```

28.5.531   rthm-seq-bar/get-nth-attack-with-tied

[rthm-seq-bar] [Methods]

DATE:

 July 24th 2015, Glenferness

DESCRIPTION:

Same as get-nth-attack method but will always return a list of the attacked
event plus any following events this is tied to.

ARGUMENTS:

- The zero-based index number indicating which attack is sought.
- The given rthm-seq-bar object in which to search.

OPTIONAL ARGUMENTS:

- T or NIL indicating whether to print a warning message if the given index
  is greater than the number of attacks in the RHYTHMS list (minus one to
  compensate for the zero-based indexing) (default = T).

RETURN VALUE:

A rhythm object.

Returns NIL if the given index is higher than the highest possible index of
attacks in the given rthm-seq-bar object.

SYNOPSIS:

```
(defmethod get-nth-attack-with-tied (index (rsb rthm-seq-bar)
                                     &optional (warn t))
```

28.5.532   rthm-seq-bar/get-nth-event

[rthm-seq-bar] [Methods]

DESCRIPTION:

Get the nth event (rhythm) in the given rthm-seq-bar object. This is a
zero-based index.

The method defaults to interrupting with an error if the n-value is greater
than the number of items in the rthm-seq-bar. This can be disabled using
the optional argument.

ARGUMENTS:

- A rthm-seq-bar object.
- An index number.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to interrupt and drop into the debugger with
   an error. Default = T.

RETURN VALUE:

 A rhythm object when successful.

 Returns NIL when the specified index number is greater than the number of
 events in the rthm-seq-bar object. Also prints an error in this case by
 default, which can be suppressed by setting the optional argument to NIL.

EXAMPLE:

```
;; Zero-based indexing. Returns a rhythm object when successful.
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
(get-nth-event 0 rsb))

=>
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
score-rthm: 4.0f0, undotted-value: 4, num-flags: 0, num-dots: 0,
is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q

;; Interrupts with an error and drops into the debugger by default if the
;; specified index number is greater than the number of events in the
;; rthm-seq-bar.
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
(get-nth-event 4 rsb))

=>
rthm-seq-bar::get-nth-event: Couldn't get event with index 4
[Condition of type SIMPLE-ERROR]

;; The error can be suppressed by setting the optional argument to NIL
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
(get-nth-event 4 rsb nil))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-event (index (rsb rthm-seq-bar)
                         &optional (error t))
```

28.5.533   rthm-seq-bar/get-nth-non-rest-rhythm

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Get the nth non-rest rhythm object stored in the given rthm-seq-bar.

ARGUMENTS:

 - The zero-based index number indicating which non-rest-rhythm is sought.
 - The given rthm-seq-bar object in which to search.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print an error message if the given index
   is greater than the number of non-rest rhythms in the RHYTHMS list (minus
   one to compensate for the zero-based indexing). (Default = T).

RETURN VALUE:

 A rhythm object.

 Returns NIL if the given index is higher than the highest possible index of
 non-rest rhythms in the given rthm-seq-bar object.

EXAMPLE:

```
;; The method returns a rhythm object when successful ; ;
(let ((rsb (make-rthm-seq-bar '((2 4) e (e) s s (s) s))))
(get-nth-non-rest-rhythm 0 rsb))

=>
RHYTHM: value: 8.0, duration: 0.5, rq: 1/2, is-rest: NIL, score-rthm: 8.0,
undotted-value: 8, num-flags: 1, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 0.5, is-grace-note: NIL,
needs-new-note: T, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 8,
tuplet-scaler: 1, grace-note-duration: 0.05,
LINKED-NAMED-OBJECT: previous: NIL
```

```
this: NIL
next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E

(let ((rsb (make-rthm-seq-bar '((2 4) e (e) s s (s) s))))
(data (get-nth-non-rest-rhythm 1 rsb)))

=> S

(let ((rsb (make-rthm-seq-bar '((2 4) e (e) s s (s) s))))
(data (get-nth-non-rest-rhythm 4 rsb)))

=>
Evaluation aborted on #<SIMPLE-ERROR>
rthm-seq-bar::get-nth-non-rest-rhythm: Couldn't get non-rest rhythm with index
4 for bar number -1
[Condition of type SIMPLE-ERROR]

(let ((rsb (make-rthm-seq-bar '((2 4) e (e) s s (s) s))))
(get-nth-non-rest-rhythm 4 rsb nil))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-non-rest-rhythm (index (rsb rthm-seq-bar)
                                      &optional (error t))
```

28.5.534   rthm-seq-bar/get-nth-rest

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Gets the rhythm object of the nth rest in a given rthm-seq-bar.

ARGUMENTS:

 - The zero-based index number indicating which rest is sought.
 - The given rthm-seq-bar object in which to search.

OPTIONAL ARGUMENTS:

 - T or NIL indicating whether to print an error message if the given index

   is greater than the number of rests in the RHYTHMS list (minus one to
   compensate for the zero-based indexing) (default = T).

RETURN VALUE:

 A rhythm object.

 Returns NIL if the given index is higher than the highest possible index of
 rests in the given rthm-seq-bar object.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((3 4) e (e) s s (s) s (q)))))
(get-nth-rest 0 rsb))

=>
RHYTHM: value: 8.0, duration: 0.5, rq: 1/2, is-rest: T, score-rthm: 8.0,
undotted-value: 8, num-flags: 1, num-dots: 0, is-tied-to: NIL,
is-tied-from: NIL, compound-duration: 0.5, is-grace-note: NIL,
needs-new-note: NIL, beam: NIL, bracket: NIL, rqq-note: NIL,
rqq-info: NIL, marks: NIL, marks-in-part: NIL, letter-value: 8,
tuplet-scaler: 1, grace-note-duration: 0.05,
LINKED-NAMED-OBJECT: previous: NIL
this: NIL
next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E

(let ((rsb (make-rthm-seq-bar '((3 4) e (e) s s (s) s (q)))))
(data (get-nth-rest 2 rsb)))

=> Q

(let ((rsb (make-rthm-seq-bar '((3 4) e (e) s s (s) s (q)))))
(get-nth-rest 3 rsb t))

Evaluation aborted on #<SIMPLE-ERROR>
rthm-seq-bar::get-nth-rest: Couldn't get rest with index 3
[Condition of type SIMPLE-ERROR]

(let ((rsb (make-rthm-seq-bar '((3 4) e (e) s s (s) s (q)))))
(get-nth-rest 3 rsb nil))

=> NIL
```

SYNOPSIS:

```
(defmethod get-nth-rest (index (rsb rthm-seq-bar)
                         &optional (error t))
```

28.5.535   rthm-seq-bar/get-pitch-symbols

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Return a list of the pitch symbols for the events in the bar.

ARGUMENTS:

 - a rthm-seq-bar object

RETURN VALUE:

 A list of pitch symbols

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :set-palette '((1 ((gs4 bf4 c4))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                :pitch-seq-palette ((1 2 1 2 1 1 3 1)))))
        :rthm-seq-map '((1 ((vn (1 1 1)))))))))
  (get-pitch-symbols (get-bar mini 1 'vn)))
  =>
  (C4 GS4 C4 GS4 C4 C4 BF4 C4)
```

SYNOPSIS:

```
(defmethod get-pitch-symbols ((rsb rthm-seq-bar) &optional written)
```

28.5.536   rthm-seq-bar/get-rhythm-symbols

[rthm-seq-bar] [Methods]
DATE:

 01-May-2012

DESCRIPTION:

 Return the rhythms of a given rthm-seq-bar object as a list of rhythm
 symbols.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 - A list of rhythm symbols.

EXAMPLE:

```
  (let ((rsb (make-rthm-seq-bar '((4 4) q e s s q. e))))
(get-rhythm-symbols rsb))

  => (Q E S S Q. E)
```

SYNOPSIS:

```
(defmethod get-rhythm-symbols ((rsb rthm-seq-bar))
```


28.5.537   rthm-seq-bar/get-time-sig

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Return the time-sig object for the given rthm-seq-bar object.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 A time-sig object.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
  (get-time-sig rsb))
```

```
=>
TIME-SIG: num: 2, denom: 4, duration: 2.0, compound: NIL, midi-clocks: 24,
num-beats: 2
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0204", tag: NIL,
data: (2 4)
```

SYNOPSIS:

```
(defmethod get-time-sig ((rsb rthm-seq-bar) &optional ignore)
```

28.5.538   rthm-seq-bar/get-time-sig-as-list

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Get the time signature for a given rthm-seq-bar object in list form.

ARGUMENTS:

 - A rthm-seq-bar object.

RETURN VALUE:

 A list.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
  (get-time-sig-as-list rsb))
=> (2 4)
```

SYNOPSIS:

```
(defmethod get-time-sig-as-list ((rsb rthm-seq-bar))
```

28.5.539   rthm-seq-bar/invert

[rthm-seq-bar] [Methods]

DATE:

 11th July 2020, Heidhausen

DESCRIPTION:

 Turn all the rests in a rthm-seq-bar into notes and all the notes/chords
 into rests. NB this method is destructive.

ARGUMENTS:

 - the rthm-seq-bar object

OPTIONAL ARGUMENTS:

 - a list of pitches/chords (either objects or symbols/lists of symbols) to
   replace the rests with. This will be used cyclically if there aren't
   enough for the number of generated rests. They will not all be used up if
   there aren't enough rests. If NIL then all rests will be replaced with
   middle Cs.
 - T or NIL to tie notes rather than use new ones when more than one
   consecutive rest is created.

RETURN VALUE:

 the modified rthm-seq-bar object

EXAMPLE:

```
(let ((r (make-rest 'e))) (setf (pitch-or-chord r) '(c4 e4)) r)
--->
EVENT: start-time: NIL, end-time: NIL,
       duration-in-tempo: 0.000,
       compound-duration-in-tempo: 0.000,
       amplitude: 0.300
       bar-num: -1, marks-before: NIL,
       tempo-change: NIL
       instrument-change: NIL
       display-tempo: NIL, start-time-qtrs: -1.000,
       midi-time-sig: NIL, midi-program-changes: NIL,
       midi-control-changes: NIL,
       8va: 0, player: NIL
       asco-label: NIL, asco-msgs: NIL
       set-ref: NIL
       pitch-or-chord:
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL, micro-tonality: 0.0
centroid: NIL, dissonance: NIL
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
```

```
                        this: NIL,
                        next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
PITCH: frequency: 261.626, midi-note: 60, midi-channel: NIL
...
```

SYNOPSIS:

```
(defmethod invert ((rsb rthm-seq-bar) &optional pitch-list tie)
```

28.5.540   rthm-seq-bar/make-rest-bar

[rthm-seq-bar] [Functions]

DESCRIPTION:

 Make a rthm-seq-bar object that consists of a bar of rest.

ARGUMENTS:

 - The time signature of the rthm-seq-bar object to be made, as a quoted
   list.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print the time signature in score output.
 - show-rest. This argument indicates whether or not to print the rest in
   the printed score output (CMN/LilyPond). Default = T.

 The remaining optional arguments are set internally by the
 slippery-chicken class, but can be read by the user for debugging.
 - missing-duration: Indicates whether the bar is missing a duration.
 - player-section-ref: The current player and section of the given
   rthm-seq-bar object.
 - nth-seq: The current sequenz (with a "z") of the given rthm-seq-bar
   object.
 - nth-bar: The current bar number of the given rthm-seq-bar object.

RETURN VALUE:

 A rthm-seq-bar object.

EXAMPLE:

```
  (let ((rsb-rb (make-rest-bar '(2 4) nil t)))
(format t "~%time-sig: ~a~%is-rest-bar: ~a~%write-time-sig: ~
             ~a~%show-rest: ~a~%"
(data (get-time-sig rsb-rb))
(is-rest-bar rsb-rb)
(write-time-sig rsb-rb)
(show-rest rsb-rb))
(print-simple rsb-rb)
rsb-rb)

  =>
  RTHM-SEQ-BAR: time-sig: 0 (2 4), time-sig-given: T, bar-num: -1,
  [...]

  time-sig: (2 4)
  is-rest-bar: T
  write-time-sig: NIL
  show-rest: T
  (2 4): rest 2,
```

SYNOPSIS:

```
(defun make-rest-bar (time-sig &optional
                                 write-time-sig
                                 (show-rest t)
                                 missing-duration
                                 player-section-ref nth-seq
                                 nth-bar)
```

28.5.541   rthm-seq-bar/make-rsb-from-unit-multipliers

[rthm-seq-bar] [Functions]

DATE:

 Jun 6th 2020, Heidhausen

DESCRIPTION:

 Make a rthm-seq-bar using multiples of a single unit e.g. '(2 2 1) would
 result in an automatic time signature of 5/8 if passed a unit of 'e (or
 8). In that case the rhythms would be q q e. Rests may be created by passing
 a number in parentheses.

ARGUMENTS:

- the rhythm unit (symbol, number or rhythm object)
- the list of multipliers

RETURN VALUE:

 a rthm-seq-bar object

EXAMPLE:

```
(print-simple (make-rsb-from-unit-multipliers 'e '(3 4 (1) 2)))
-->
NIL: bar -1: (10 8):
note Q.,
note H,
rest E,
note Q,
```

SYNOPSIS:

```
(defun make-rsb-from-unit-multipliers (unit multipliers)
```

28.5.542   rthm-seq-bar/make-rthm-seq-bar

[rthm-seq-bar] [Functions]

DESCRIPTION:

 Public interface for creating a rthm-seq-bar object, each instance of which
 which holds one of the individual bars that reside in a rhythmic
 sequence.

 This class is responsible for parsing lists containing rhythms and time
 signatures, but not for parsing these things themselves--that is done by
 separate classes.

 A { followed by a number means that all the notes from now to the } will be
 enclosed in a bracket with the number inside. This may be nested. A -
 indicates beaming: the first - indicates the start of a beam, the second
 the end of that beam.

ARGUMENTS:

- A list of rhythmic durations, which may include ties and dots. Durations
  may be written as numeric (integer) values or may use the CM/CMN/SCORE
  alphabetic shorthand s=16, e=8, q=4, h=2, w=1. NB: Repeating rhythms can

  be indicated using a shorthand notation consisting of a multiplication
  symbol ('x'), e.g.: (make-rthm-seq-bar '((4 4) s x 16)).

make-rthm-seq-bar requires a time signature. If no time signature is
provided, the most recently defined time signature will be used. If one is
provided, it must be included as the first element of the data list. The
time signature is formulated as a list containing two integers, the first
being the number of beats in the bar and the second being the beat unit for
the bar.

OPTIONAL ARGUMENTS:

 - A name (symbol) for the object ID.

RETURN VALUE:

 Returns a rthm-seq-bar.

EXAMPLE:

```
  (make-rthm-seq-bar '((2 4) q e s s))

  =>
 RTHM-SEQ-BAR:
 [...]
 NAMED-OBJECT: id: NIL, tag: NIL,
 data: ((2 4) Q E S S)

  (make-rthm-seq-bar '((2 4) q e s s) 'test)
  =>
 RTHM-SEQ-BAR:
 [...]
 NAMED-OBJECT: id: TEST, tag: NIL,
 data: ((2 4) Q E S S)

  (make-rthm-seq-bar '((2 4) q \+16\.+32 e))
  =>
 RTHM-SEQ-BAR:
 [...]
 NAMED-OBJECT: id: NIL, tag: NIL,
 data: ((2 4) Q +16.+32 E)

  (make-rthm-seq-bar '((2 4) { 3 te te te } q))
  =>
 RTHM-SEQ-BAR:
```

```
 [...]
 NAMED-OBJECT: id: NIL, tag: NIL,
 data: ((2 4) { 3 TE TE TE } Q)
```

SYNOPSIS:

```
(defun make-rthm-seq-bar (rhythms &optional name)
```

28.5.543   rthm-seq-bar/remove-dynamics

[rthm-seq-bar] [Methods]

DATE:

 June 13th

DESCRIPTION:

 remove all dynamics attached to events in the bar

ARGUMENTS:

 - a rthm-seq-bar object

OPTIONAL ARGUMENTS:

 - T or NIL to remove hairpin marks too (e.g. cresc-beg). Default = NIL.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod remove-dynamics ((rsb rthm-seq-bar) &optional hairpins)
```

28.5.544   rthm-seq-bar/remove-rhythms

[rthm-seq-bar] [Methods]

DATE:

 February 8th 2017

DESCRIPTION:

 remove a specified number of rhythms/events from a rthm-seq-bar object.

ARGUMENTS:

 - the rthm-seq-bar object
 - the rhythm/event to start removing at (1-based)
 - how many rhythms/events to remove

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether the tuplets slot of the rthm-seq-bar should
   be updated to reflect changes. Default = T.
 - T or NIL to indicated whether the bracket slot of the starting
   rhythm/event should take the value of the last rhythm/event removed

RETURN VALUE:

 the list of rhythm/event objects from the rhythms slot of the rthm-seq-bar

SYNOPSIS:

```
(defmethod remove-rhythms ((rsb rthm-seq-bar) start how-many
                           &optional (update-tuplets t)
                               (inherit-last-tuplet t))
```

28.5.545   rthm-seq-bar/replace-rhythms

[rthm-seq-bar] [Methods]

DATE:

 February 16th 2017, Edinburgh

DESCRIPTION:

 Replace the rhythms/events in a rthm-seq-bar object, with an option to
 automatically beam the result. The bar is checked to make sure it is full
 after replacing and an error will be issued if it is not.

ARGUMENTS:

 - a rthm-seq-bar object
 - the index (1-based) of the rhythm/events to start at (integer)
 - the number of rhythms/events to replace (integer)
 - a list of the new rhythm/event objects

OPTIONAL ARGUMENTS:

```
T or NIL to indicate whether the automatic beaming routine should be called
after the replacement
```

RETURN VALUE:

```
What the is-full method returns: two values: T or NIL to indicate whether
the bar is full, and the difference between the duration of the time
signature and the rhythms (usually a small floating-point error difference)
```

SYNOPSIS:

```
(defmethod replace-rhythms ((rsb rthm-seq-bar) start-rhythm replace-num-rhythms
                            new-rhythms &optional auto-beam)
```

28.5.546   rthm-seq-bar/reset-8va

[rthm-seq-bar] [Methods]

DATE:

```
22 Sep 2011
```

DESCRIPTION:

```
Reset the 8VA slots of all event objects within a given rthm-seq-object to
0 (no ottava/ottava bassa transposition).
```

ARGUMENTS:

```
- A rthm-seq-bar object
```

RETURN VALUE:

```
Always returns NIL.
```

EXAMPLE:

```
;; Create a rthm-seq-bar object consisting of event objects, print the default ; ; ; ; ; ;
;; value of the 8VA slots for those events. Set the 8VA slots to 1 and print ; ; ; ; ; ;
;; the value of those slots to see the change. Apply the reset-8va method to ; ; ; ; ; ;
;; remove any values and reset the slots to NIL, and print the results. ; ; ; ; ; ;
```

```
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(print (loop for e in (rhythms rsb) collect (8va e)))
(set-8va rsb 1)
(print (loop for e in (rhythms rsb) collect (8va e)))
(reset-8va rsb)
(print (loop for e in (rhythms rsb) collect (8va e))))

  =>
  (0 0 0)
  (1 1 1)
  (0 0 0)
```

SYNOPSIS:

```
(defmethod reset-8va ((rsb rthm-seq-bar))
```

28.5.547   rthm-seq-bar/respell-bar

[rthm-seq-bar] [Methods]

DESCRIPTION:

Scan the specified rthm-seq-bar object for enharmonically equivalent
pitches and unify their spelling. This method won't generally be called by
the user directly, rather, it's called by the respell-bars method in the
slippery-chicken class.

Clearly, this method will not work if the rthm-seq-bar only contains rhythm
objects: it's made to be called when these have been promoted to event
objects during the initialization of a slippery-chicken object.

NB: Although this method focuses on just one rthm-seq-bar object, the
    parent slippery-chicken object and player ID are needed in order to
    determine ties that may exist into the next bar from the present bar.

NB: The slippery-chicken class version of the method of this name uses
    pitches from previous bars as well when respelling a given rthm-seq-bar
    object, so different results are normal. Users should generally call
    the slippery-chicken method rather than calling this one directly on
    individual bars.

ARGUMENTS:

- A rthm-seq-bar object.
- A slippery-chicken object.
- A player ID (symbol).

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to process only written or only sounding
  pitches. T = only written. Default = NIL.
- The last attack (event object) of the previous bar. This is usually
  supplied by the calling method. Default = NIL.

RETURN VALUE:

 Returns the rthm-seq-bar object it was passed.

EXAMPLE:

```
;;; Create a slippery-chicken object using pitches GS4 and AF4, print the ; ; ; ; ; ;
;;; pitches of a specified bar within that object. Apply respell-bar and print ; ; ; ; ; ;
;;; the same pitches again to see the difference. ; ; ; ; ; ;

  (let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((vn (violin :midi-channel 1))))
:set-palette '((1 ((gs4 af4 bf4))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
:pitch-seq-palette ((1 2 1 1 1 1 1 1)))))
:rthm-seq-map '((1 ((vn (1 1 1)))))))))
(print (loop for r in (rhythms (get-bar mini 2 'vn))
collect (get-pitch-symbol r)))
(respell-bar (get-bar mini 2 'vn) mini 'vn)
(print (loop for r in (rhythms (get-bar mini 2 'vn))
collect (get-pitch-symbol r))))
  =>
  (GS4 AF4 GS4 GS4 GS4 GS4 GS4 GS4)
  (GS4 GS4 GS4 GS4 GS4 GS4 GS4 GS4)
```

SYNOPSIS:

```
(defmethod respell-bar ((rsb rthm-seq-bar) sc player
                        &optional written last-attack-previous-bar)
```

28.5.548   rthm-seq-bar/scale

[rthm-seq-bar] [Methods]

DESCRIPTION:

Change the values of a rthm-seq-bar objects rhythm durations by a specified
scaling factor.

This method always returns a new rthm-seq-bar object, recreating scaled
rhythms with beams etc. where appropriate. See time-sig::scale for details
on how the new meter is created.

ARGUMENTS:

 - A rthm-seq-bar object.
 - A number that is the scaling factor.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to preserve the original meter (duple,
   triple, quadruple etc.)
 - (two ignore arguments for internal use only)

RETURN VALUE:

 Returns a rthm-seq-bar object

EXAMPLE:

```
;;; Create a rthm-seq-bar object and scale its durations by a fact of
;;; 2. Returns a rthm-seq-bar object.
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
(scale rsb 2))

=>
RTHM-SEQ-BAR: time-sig: 19 (2 2), time-sig-given: T, bar-num: -1,
[...]
RHYTHM: value: 2.000, duration: 2.000, rq: 2, is-rest: NIL,
[...]
data: H
[...]
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
[...]
data: Q
```

```
[...]
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: NIL,
[...]
data: E
[...]
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: NIL,
[...]
data: E
[...]

;;; Use the print-simple method to see formatted results
(let ((rsb (make-rthm-seq-bar '((2 4) q e s s))))
(print-simple (scale rsb .5)))

=>
(2 8): note E, note S, note 32, note 32,

;;; Set the optional <preserve-meter> argument to NIL to allow the method to
;;; return results in a different metric quality (this returns a quadruple
;;; meter rather than a duple)
(let ((rsb (make-rthm-seq-bar '((6 8) q e q s s))))
(print-simple (scale rsb 2 nil)))

=>
(12 8): note H, note Q, note H, note E, note E,
```

SYNOPSIS:

```
(defmethod scale ((rsb rthm-seq-bar) scaler
                  &optional (preserve-meter t) ignore1 ignore2)
```

28.5.549   rthm-seq-bar/set-8va

[rthm-seq-bar] [Methods]

DATE:

 23-Sep-2011

DESCRIPTION:

 Set the 8VA (ottava) slots of the event objects within a given rthm-seq-bar
 object. This number can be positive or negative. Only the values 1, 0 and
 -1 are valid for the number of octaves to be transposed.

ARGUMENTS:

- A rthm-seq-bar object.
- A number indicating the number of octaves to be transposed in either
  direction (ottava/ottava bassa).

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; The method returns NIL              ; ; ; ; ;

  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-8va rsb 1))

  => NIL

;; Create a rthm-seq-bar object with event objects, set the 8va slot to 1, and ; ; ; ; ;
;; access and print it to see it's new value. ; ; ; ; ;

  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-8va rsb 1)
(loop for e in (rhythms rsb) collect (8va e)))

  => (1 1 1)
```

SYNOPSIS:

```
(defmethod set-8va ((rsb rthm-seq-bar) 8va)
```

28.5.550   rthm-seq-bar/set-amplitudes

[rthm-seq-bar] [Methods]

DESCRIPTION:

Add a specified amplitude (between 0.0 and 1.0) to all non-rest event
objects in a specified rthm-seq-bar object.

ARGUMENTS:

 - A rthm-seq-bar object.
 - A number that is an amplitude value between 0.0 and 1.0.

RETURN VALUE:

 Returns the amplitude value set.

EXAMPLE:

```
  (let* ((mini
(make-slippery-chicken
'+sc-object+
:ensemble '(((va (viola :midi-channel 2))))
:set-palette '((1 ((c3 d3 e3 f3 g3 a3 b3 c4))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((4 4) e (e) e (e) (e) e e e))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((va (1 1 1)))))))))
(set-amplitudes (get-bar mini 2 'va) 0.9)
(cmn-display mini))
```

SYNOPSIS:

```
(defmethod set-amplitudes ((rsb rthm-seq-bar) amp)
```

28.5.551   rthm-seq-bar/set-dynamics

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Add a specified dynamic mark to all attacked event objects (i.e. not rests
 and not notes that are tied to) in a specified rthm-seq-bar-object. This
 method was created mainly to make it easy to set amplitudes for a range of
 notes (e.g. with map-over-bars), so that they are, for example, reflected
 in MIDI velocities. If used over many notes the score will probably then be
 littered with extraneous dynamic marks. These can then be removed, if so
 desired, with the slippery-chicken class remove-extraneous-dynamics method.

ARGUMENTS:

- A rthm-seq-bar object.
- A dynamic mark.

RETURN VALUE:

The specified dynamic mark

EXAMPLE:

```
  (let* ((mini
(make-slippery-chicken
'+sc-object+
:ensemble '(((va (viola :midi-channel 2))))
:set-palette '((1 ((c3 d3 e3 f3 g3 a3 b3 c4))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((4 4) e (e) e (e) (e) e e e))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((va (1 1 1)))))))))
(set-dynamics (get-bar mini 2 'va) 'ppp))

  => PPP
```

SYNOPSIS:

```
(defmethod set-dynamics ((rsb rthm-seq-bar) dynamic)
```

28.5.552   rthm-seq-bar/set-midi-channel

[rthm-seq-bar] [Methods]

DESCRIPTION:

Set the MIDI-channel and microtonal MIDI-channel for the pitch object
of an event object within a given rthm-seq-bar object. Sets the
MIDI-CHANNEL slot of all event objects contained in the rthm-seq-bar object
to the same channel.

ARGUMENTS:

- A rthm-seq-bar object.
- A whole number indicating the MIDI channel to be used for the
  equal-tempered pitch material of the given rthm-seq-bar object.
- A whole number indicating the MIDI channel to be used for microtonal
  pitch material of the given rthm-seq-bar object.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Create a rthm-seq-bar using event objects and check the MIDI-CHANNEL slots ; ; ; ; ; ;
;; of those event objects to see that they are NIL by default. ; ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(loop for p in (rhythms rsb)
collect (midi-channel (pitch-or-chord p))))

  => (NIL NIL NIL)

;; Apply the set-midi-channel method to the rthm-seq-bar object and read and ; ; ; ; ; ;
;; print the MIDI-CHANNEL slots of each of the individual events to see that ; ; ; ; ; ;
;; they've been set.                      ; ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-midi-channel rsb 13 14)
(loop for p in (rhythms rsb)
collect (midi-channel (pitch-or-chord p))))

  => (13 13 13)
```

SYNOPSIS:

```
(defmethod set-midi-channel ((rsb rthm-seq-bar) midi-channel
                             &optional microtonal-midi-channel)
```

28.5.553   rthm-seq-bar/set-nth-attack

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Sets the value of the nth rhythm object of a given rthm-seq-bar that needs

an attack; i.e., not a rest and not a tied note.

NB: This method does not check to ensure that the resulting rthm-seq-bar
contains the right number of beats.

ARGUMENTS:

- A zero-based index number for the attacked note to change.
- An event.
- A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

- T or NIL indicating whether to print a warning message if the given index
  (minus one) is greater than the number of attacks in the RHYTHMS
  list. Default = T.

RETURN VALUE:

An event object.

EXAMPLE:

```
(let ((rsb (make-rthm-seq-bar '((2 4) q+e s s))))
(set-nth-attack 1 (make-event 'e4 'q) rsb))

=>
EVENT: start-time: NIL, end-time: NIL,
[...]
PITCH: frequency: 329.6275526703903d0, midi-note: 64, midi-channel: NIL
[...]
NAMED-OBJECT: id: E4, tag: NIL,
data: E4
[...]
RHYTHM: value: 4.0, duration: 1.0, rq: 1, is-rest: NIL, score-rthm: 4.0,
[...]
NAMED-OBJECT: id: Q, tag: NIL,
data: Q

(let ((rsb (make-rthm-seq-bar '((2 4) q+e s s))))
(set-nth-attack 2 (make-event 'e4 'q) rsb)
(loop for r in (rhythms rsb) collect (data r)))

=> ("Q" "E" S Q)
```

```
(let ((rsb (make-rthm-seq-bar '((2 4) q+e s s))))
(set-nth-attack 3 (make-event 'e4 'q) rsb))

=> NIL
rthm-seq-bar::set-nth-attack: index (3) < 0 or >= notes-needed (3)

(let ((rsb (make-rthm-seq-bar '((2 4) q+e s s))))
(set-nth-attack 3 (make-event 'e4 'q) rsb nil))

=> NIL
```

SYNOPSIS:

```
(defmethod set-nth-attack (index (e event) (rsb rthm-seq-bar)
                           &optional (warn t))
```

28.5.554   rthm-seq-bar/set-written

[rthm-seq-bar] [Methods]

DATE:

 20 Jul 2011 (Pula)

DESCRIPTION:

 Set the written pitch (as opposed to sounding; i.e., for transposing
 instruments) of an event object within a given rthm-seq-bar object. The
 sounding pitch remains unchanged as a pitch object in the PITCH-OR-CHORD
 slot, while the written pitch is added as a pitch object to the
 WRITTEN-PITCH-OR-CHORD slot.

ARGUMENTS:

 - A rthm-seq-bar-object
 - A number (positive or negative) indicating the transposition by
   semitones. See this method in the event class for more information and
   examples.

RETURN VALUE:

 Always returns T.

EXAMPLE:

```
;; The method returns NIL                ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-written rsb -2))

  => T
```

```
;; Set the written pitch transposition to 2 semitones lower, then check the ; ; ; ; ;
;; data of the WRITTEN-PITCH-OR-CHORD slot of each event to see the ; ; ; ; ;
;; corresponding pitches                ; ; ; ; ;
  (let ((rsb (make-rthm-seq-bar
(list
'(3 8)
(make-event 'cs4 'e)
(make-event 'cs4 'e)
(make-event 'cs4 'e)))))
(set-written rsb -2)
(loop for p in (rhythms rsb)
collect (get-pitch-symbol p)))

  => (B3 B3 B3)
```

SYNOPSIS:

```
(defmethod set-written ((rsb rthm-seq-bar) transposition)
```

28.5.555   rthm-seq-bar/split

[rthm-seq-bar] [Methods]

DATE:

 27 Jan 2011

DESCRIPTION:

 Splits a given rthm-seq-bar into multiple smaller rthm-seq-bar
 objects. This will only work if the given rthm-seq-bar object can be split
 into whole beats; e.g. a 4/4 bar will not be split into 5/8 + 3/8.

 The keyword arguments :min-beats and :max-beats serve as guidelines rather

than strict cut-offs. In some cases, the method may only be able to
effectively split the given rthm-seq-bar by dividing it into segments that
exceed the length stipulated by these arguments (see example below).

Depending on the min-beats/max-beats arguments stipulated by the user or
the rhythmic structure of the given rthm-seq-bar object, the given
rthm-seq-bar may not be splittable, in which case NIL is returned. If the
keyword argument :warn is set to T, a warning will be also be printed in
such cases.

NB The method does not copy over and update bar start-times (this is meant
to be done at the rthm-seq stage, not once the whole piece has been
generated).

ARGUMENTS:

 - A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :min-beats. This argument takes an integer value to indicate the minimum
  number of beats in any of the new rthm-seq-bar objects created. This
  serves as a guideline only and may occasionally be exceeded in value by
  the method. Default value = 2.
- :max-beats. This argument takes an integer value to indicate the maximum
  number of beats in any of the new rthm-seq-bar objects created. This
  serves as a guideline only and may occasionally be exceeded in value by
  the method. Default value = 5.
- :warn. Indicates whether to print a warning if the rthm-seq-bar object is
  unsplittable. Value T = print a warning. Defaults to NIL.

RETURN VALUE:

 Returns a list of rthm-seq-bar objects if successful, NIL if not.

EXAMPLE:

```
  (let* ((rsb (make-rthm-seq-bar '((7 4) h. e e +e. e. e q)))
(rsb-splt (split rsb)))
(loop for i in rsb-splt collect
(loop for r in (rhythms i) collect (data r))))

  => ((H.) (E E "E." E. E Q))
```

```
  (let* ((rsb (make-rthm-seq-bar '((7 4) h. e e +e. e. e q)))
(rsb-splt (split rsb)))
(loop for i in rsb-splt do (print-simple i)))

  =>
  (3 4): note H.,
  (4 4): note E, note E, note E., note E., note E, note Q,

  (let* ((rsb (make-rthm-seq-bar '((7 4) h. e e +e. e. e q)))
(rsb-splt (split rsb :min-beats 1 :max-beats 3)))
(loop for i in rsb-splt do (print-simple i)))

  =>
  (3 4): note H.,
  (1 4): note E, note E,
  (2 4): note E., note E., note E,
  (1 4): note Q,

  (let ((rsb (make-rthm-seq-bar '((7 4) h. e e +e. e. e q))))
(split rsb :max-beats 1 :warn t))

  => NIL
  WARNING: rthm-seq-bar::split: couldn't split bar:
```

SYNOPSIS:

```
(defmethod split ((rsb rthm-seq-bar) &key
                                        (min-beats 2) (max-beats 5) warn ignore)
```

28.5.556   rthm-seq-bar/time-sig-equal

[rthm-seq-bar] [Methods]

DESCRIPTION:

Check to see if two given rthm-seq-bar objects have the same time signature.

ARGUMENTS:

 - Two rthm-seq-bar objects.

RETURN VALUE:

 T if the given rthm-seq-bar objects have the same time signature.
 NIL if the given rthm-seq-bar objects have different times signatures.

EXAMPLE:

```
(let ((rsb1 (make-rthm-seq-bar '((2 4) q e s s)))
      (rsb2 (make-rthm-seq-bar '((2 4) s s e q))))
  (time-sig-equal rsb1 rsb2))

=> T

(let ((rsb1 (make-rthm-seq-bar '((2 4) q e s s)))
      (rsb2 (make-rthm-seq-bar '((3 4) q+e e s s s s))))
  (time-sig-equal rsb1 rsb2))
=> NIL
```

SYNOPSIS:

```
(defmethod time-sig-equal ((rsb1 rthm-seq-bar) (rsb2 rthm-seq-bar))
```

28.5.557   rthm-seq-bar/transpose

[rthm-seq-bar] [Methods]

DESCRIPTION:

 Transpose the pitches of event objects stored in a rthm-seq-bar object by a
 specified number of semitones (positive for up, negative for down).

ARGUMENTS:

 - A rthm-seq-bar object.
 - A whole number (positive or negative).

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :destructively. Set to T or NIL to indicate whether the slot values of
   the original rthm-seq-bar object should be changed or not (even though
   the method always returns a clone). T = change the originals.
   Default = NIL.
 - :chord-function. A function to be used for the transposition of
   chords. Default = #'transpose.
 - :pitch-function. A function to be used for the transposition of
   pitches. Default = #'transpose.

RETURN VALUE:

This method returns a clone of the rthm-seq-bar object whether the keyword
argument :destructively is set to T or NIL. It does change the
corresponding slot values of the original when set to T even though it
returns the clone.

EXAMPLE:

```
;; Create a rthm-seq-bar object using make-event, transpose the contained
;; pitches destructively, and read the values of the corresponding slots to see
;; the change.
(let ((rsb (make-rthm-seq-bar
            (list
             '(3 8)
             (make-event 'cs4 'e)
             (make-event 'cs4 'e)
             (make-event 'cs4 'e)))))
  (transpose rsb 3 :destructively t)
  (loop for p in (rhythms rsb)
     collect (data (pitch-or-chord p))))

=> (E4 E4 E4)

;; Do the same thing without the :destructively keyword being set to T
(let ((rsb (make-rthm-seq-bar
            (list
             '(3 8)
             (make-event 'cs4 'e)
             (make-event 'cs4 'e)
             (make-event 'cs4 'e)))))
  (transpose rsb 3)
  (loop for p in (rhythms rsb)
     collect (data (pitch-or-chord p))))

=> (CS4 CS4 CS4)
```

SYNOPSIS:

```
(defmethod transpose ((rsb rthm-seq-bar) semitones
                      &key
                      ;; when t, then the events will be replaced by the
                      ;; transposition.
                      (destructively nil)
                      ;; the default functions are the class methods for pitch
                      ;; or chord.
                      (chord-function #'transpose)
                      (pitch-function #'transpose))
```

28.5.558   sclist/sc-nthcdr

[sclist] [Methods]

DESCRIPTION:

Get the tail (rest) of a given sclist object beginning with and including
the specified zero-based index number.

NB: This method is destructive and replaces the contents of the given list
with the sublist returned by the method.

ARGUMENTS:

- An index number.
- An sclist object

RETURN VALUE:

Returns a list.

Returns NIL if the specified index is greater (minus 1) than the number of
items in the given list.

EXAMPLE:

```
;; Create an sclist object and get the tail of the list starting at place
;; 4. The subset returned replaces the data of the original.
(let ((scl (make-sclist '(0 1 2 3 4 5 6 7 8 9))))
  (sc-nthcdr 4 scl)
  (data scl))

=> (4 5 6 7 8 9)

(let ((scl (make-sclist '(0 1 2 3 4 5 6 7 8 9))))
  (sc-nthcdr 14 scl))

=> NIL
```

SYNOPSIS:

```
(defmethod sc-nthcdr (nth (scl sclist))
```

28.5.559   sclist/sc-set

[sclist] [Classes]

NAME:

player

 File: sc-set.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist -> chord ->
                 sc-set

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the sc-set class which holds pitch set
                 information for harmonic and pitch manipulation.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: August 10th 2001

 $$ Last modified: 14:58:36 Fri Sep 25 2020 CEST

 SVN ID: Id

28.5.560   sc-set/add

[sc-set] [Methods]

DESCRIPTION:

 Create a new sc-set object from the data of two other specified sc-set
 objects.

 NB: Any subsets contained in the original sc-set objects are lost in the
     process.

ARGUMENTS:

 - A first sc-set object.
 - A second sc-set object.

OPTIONAL ARGUMENTS:

 (- optional argument <ignore> is internal only)

RETURN VALUE: EXAMPLE:

```
(let ((mscs1 (make-sc-set '(d2 a2 e3 b3 gf4 df5 af5)))
      (mscs2 (make-sc-set '(f2 c3 g3 d4 bf4 f5 c6))))
  (add mscs1 mscs2))

=>
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 0
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 0, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: USED-NOTES, tag: NIL,
data: NIL

**** N.B. All pitches printed as symbols only, internally they are all
pitch-objects.

    subsets:
    related-sets:
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5 C6)
```

SYNOPSIS:

```
(defmethod add ((s1 sc-set) (s2 sc-set) &optional ignore)
```

28.5.561   sc-set/add-harmonics

[sc-set] [Methods]

DESCRIPTION:

 Adds pitches to the set which are harmonically related to the existing
 pitches. The keywords are the same as for the get-harmonics function. NB
 This will automatically sort all pitches from high to low (irrespective of
 the auto-sort slot).

ARGUMENTS:

 - an sc-set object

OPTIONAL ARGUMENTS:

```
keyword arguments:
  see get-harmonics function
```

RETURN VALUE:

the same set object as the first argument but with new pitches added.

EXAMPLE:

```
;;; treat the existing pitches as fundamentals. Note that the existence of E6
;;; twice in the result is not a mistake: though E6 is the nearest pitch, the
;;; frequencies are different.
(let ((s (make-sc-set '(c4 e4) :id 'test)))
  (add-harmonics s :start-partial 3 :max-results 3))
=>
SC-SET: auto-sort: T, rm-dups: T, warn-dups: T used-notes:
[...]
data: (C4 E4 G5 B5 C6 E6 E6 AF6)

;;; treat the existing pitches as partials and add the fundamentals and
;;; harmonics
(let ((s (make-sc-set '(c4 e4) :id 'test)))
  (add-harmonics s :start-freq-is-partial 3 :max-results 3))
SC-SET: auto-sort: T, rm-dups: T, warn-dups: T used-notes:
[...]
data: (F2 A2 F3 A3 C4 E4)
```

SYNOPSIS:

```
(defmethod add-harmonics ((s sc-set) &rest keywords)
```

28.5.562   sc-set/average-pitch

[sc-set] [Methods]

DATE:

July 24th 2015, Glenferness

DESCRIPTION:

Calculate the average frequency of all the pitches in the chord and return
this as a new pitch-object.

ARGUMENTS:

- the sc-set object

RETURN VALUE:

a pitch-object

EXAMPLE:

```
(average-pitch (make-complete-set '(c4 cs4  d4 e4 ef4 g4 b4)))
=>
PITCH: frequency: 337.015, midi-note: 64, midi-channel: 0
      pitch-bend: 0.38
      degree: 64, data-consistent: T, white-note: E4
      nearest-chromatic: E4
      src: 1.2881585030063512d0, src-ref-pitch: C4, score-note: E4
      qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
      micro-tone: T,
      sharp: NIL, flat: NIL, natural: T,
      octave: 4, c5ths: 0, no-8ve: E, no-8ve-no-acc: E
      show-accidental: T, white-degree: 37,
      accidental: N,
      accidental-in-parentheses: NIL, marks: NIL,
      marks-before: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E4, tag: NIL,
data: E4
```

SYNOPSIS:

```
(defmethod average-pitch ((s sc-set))
```

28.5.563   sc-set/create-chord

[sc-set] [Methods]

DESCRIPTION:

Create a chord object from the pitches of the given sc-set object. As of
1.0.6 sc-set is a subset of chord instead of sc-list so this shouldn't be
necessary, but it's still here for legacy code purposes.

ARGUMENTS:

- An sc-set object.

RETURN VALUE:

 A chord object.

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 c3 d4 df5 c6))))
  (create-chord mscs))

=>
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL
SCLIST: sclist-length: 5, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
PITCH: frequency: 73.416, midi-note: 38, midi-channel: 0
[...]
)
```

SYNOPSIS:

```
(defmethod create-chord ((s sc-set))
```

28.5.564   sc-set/create-event

[sc-set] [Methods]

DESCRIPTION:

 Create an event object (that is a chord) from a given sc-set object,
 specifying a rhythmic value and a start-time (in seconds).

ARGUMENTS:

- An sc-set object.
- A rhythmic unit, either as a numerical value (32, 16 etc) or a symbol
  that is an alphabetic shorthand ('e, 's etc).
- A number that is the start time in seconds.

OPTIONAL ARGUMENTS:

- A number that is the start-time in quarter-notes rather than seconds (see
  event class documentation for more details)

RETURN VALUE:

 An event object.

EXAMPLE:

```
;; Create an event from the specified sc-set object that is a quarter-note
;; chord starting at 0.0 seconds
(let ((mscs (make-sc-set '(d2 c3 d4 df5 c6))))
  (create-event mscs 'q 0.0))

=>
EVENT: start-time: 0.000, end-time: NIL,
       duration-in-tempo: 0.000,
       compound-duration-in-tempo: 0.000,
       amplitude: 0.700
       bar-num: -1, marks-before: NIL,
       tempo-change: NIL
       instrument-change: NIL
       display-tempo: NIL, start-time-qtrs: 0.000,
       midi-time-sig: NIL, midi-program-changes: NIL,
       8va: 0
       pitch-or-chord:
CHORD: auto-sort: T, marks: NIL, micro-tone: NIL
SCLIST: sclist-length: 5, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (
PITCH: frequency: 73.416, midi-note: 38, midi-channel: 0
[...]
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
        score-rthm: 4.0f0, undotted-value: 4, num-flags: 0, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: Q, tag: NIL,
data: Q
```

SYNOPSIS:

```
(defmethod create-event ((s sc-set) rhythm start-time
                         &optional start-time-qtrs)
```

28.5.565   sc-set/force-micro-tone

[sc-set] [Methods]

DESCRIPTION:

 Change the value of the MICRO-TONE slot of all pitch objects in a given
 sc-set object to the specified <value>.

 NB If the pitches are microtonal and thus have associated pitch-bends and
 microtonal frequencies, these will not be changed, i.e. only the micro-tone
 slot is changed.

 NB: Although the MICRO-TONE slot is generally used as a boolean, this
     method allows the user to force-set it to any value.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

 - An item of any type that is to be the new value of the MICRO-TONE slot of
   all pitch objects in the given sc-set object (generally T or
   NIL). Default = NIL.

RETURN VALUE:

 Always returns NIL.

EXAMPLE:

```
;; Create an sc-set object that contains micro-tones and print the MICRO-TONE
;; slot of all of the contained pitch objects to see their values:
(let ((mscs (make-sc-set '(d2 cqs3 fs3 cs4 e4 c5 aqf5 ef6))))
  (loop for p in (data mscs) do (print (micro-tone p))))

=>
NIL
T
NIL
NIL
NIL
NIL
T
```

```
NIL

;; Now apply the force-micro-tone method to the same set using the default
;; value of NIL and print the results

(let ((mscs (make-sc-set '(d2 cqs3 fs3 cs4 e4 c5 aqf5 ef6))))
  (force-micro-tone mscs)
    (loop for p in (data mscs) do (print (micro-tone p))))

=>
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL

;; Using the same sc-set, force all the values to T
(let ((mscs (make-sc-set '(d2 cqs3 fs3 cs4 e4 c5 aqf5 ef6))))
  (force-micro-tone mscs 't)
    (loop for p in (data mscs) do (print (micro-tone p))))

=>
T
T
T
T
T
T
T
T
```

SYNOPSIS:

```
(defmethod force-micro-tone ((s sc-set) &optional value)
```

28.5.566   sc-set/get-chromatic

[sc-set] [Methods]

DESCRIPTION:

 Return those notes of a given sc-set object that are normal chromatic notes

 (i.e. no microtones).

 If a number is given for the <octave> argument, the method will transpose
 all returned pitches into the specified octave, in which case any duplicate
 pitches are removed.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :octave. NIL or an integer that is the octave designator to which all
   resulting pitches are to be transposed (i.e. the "4" in "C4" etc.)
   Default = NIL.
 - :remove-duplicates. T or NIL to indicate whether any duplicate pitches
   within an octave that are created by use of the :octave keyword argument
   are to be removed. T = remove duplicates. Default = NIL.
 - :as-symbols. T or NIL to indicate whether to return the results of the
   method as a list of note-name symbols rather than a list of pitch
   objects. T = return as note-name symbols. Default = NIL.
 - :package. The package in which the pitches are to be handled.
   Default = :sc.
 - :invert. Get the micro-tone pitches instead.

RETURN VALUE:

 Returns a list of pitch objects by default.

 When the :as-symbols argument is set to T, a list of note-name symbols is
 returned instead.

EXAMPLE:

```
;;; Returns a list of pitch objects by default
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-chromatic mscs))

=>
(
PITCH: frequency: 73.416, midi-note: 38, midi-channel: 0
       pitch-bend: 0.0
       degree: 76, data-consistent: T, white-note: D2
       nearest-chromatic: D2
```

```
        src: 0.28061550855636597, src-ref-pitch: C4, score-note: D2
        qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
        micro-tone: NIL,
        sharp: NIL, flat: NIL, natural: T,
        octave: 2, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
        show-accidental: T, white-degree: 15,
        accidental: N,
        accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: D2, tag: NIL,
data: D2

PITCH: frequency: 184.997, midi-note: 54, midi-channel: 0
[...]
)

;; Setting the :as-symbols argument to T returns a list of note-name symbols
;; instead
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-chromatic mscs
                 :as-symbols t))

=> (D2 FS3 CS4 E4 C5 AF5 D6)

;; Giving an integer as the :octave argument transposes all returned  pitches
;; to the specified octave, removing any duplicates by default.
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-chromatic mscs
                 :as-symbols t
                 :octave 4))

=> (FS4 CS4 E4 C4 AF4 D4)

;; Setting the :invert argument to T returns the non-chromatic elements of the
;; given sc-set object instead
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-chromatic mscs
                 :as-symbols t
                 :invert t))

=> (CQS3 GQF3 FQS4 BQF5)
```

SYNOPSIS:

```
(defmethod get-chromatic ((s sc-set)
                          &key
```

```
                         (octave nil)
                         (remove-duplicates t) ;; only if octave!
                         (as-symbols nil)
                         (package :sc)
                         (invert nil))
```

28.5.567   sc-set/get-degrees

[sc-set] [Methods]

DESCRIPTION:

Return the pitches contained in the given sc-set object as a list of
DEGREES (which default to quarter-tones in slippery chicken).

ARGUMENTS:

 - An sc-set object.

RETURN VALUE:

 A list of integers.

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (get-degrees mscs))
```

=> (76 82 90 96 104 110 118 124 132 140 146 154 160 168)

SYNOPSIS:

```
(defmethod get-degrees ((s sc-set))
```

28.5.568   sc-set/get-freqs

[sc-set] [Methods]

DESCRIPTION:

Return the pitches of a given sc-set object as a list of Hz frequencies

ARGUMENTS:

 - An sc-set object.

RETURN VALUE:

 A list of numbers

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (get-freqs mscs))
```

```
=> (73.41618871368837 87.30705289160142 109.99999810639679 130.8127784729004
    164.81377633519514 195.99771591817216 246.94163930037348 293.6647548547535
    369.99440456398133 466.1637395092839 554.3652698843016 698.4564231328113
    830.6093584209975 1046.5022277832031)
```

SYNOPSIS:

```
(defmethod get-freqs ((s sc-set))
```

28.5.569   sc-set/get-midi

[sc-set] [Methods]

DESCRIPTION:

 Return the pitches of a given sc-set object as a list of their equivalent
 MIDI note numbers.

ARGUMENTS:

 - An sc-set object.

RETURN VALUE:

 A list of numbers

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (get-midi mscs))
```

```
=> (38 41 45 48 52 55 59 62 66 70 73 77 80 84)
```

SYNOPSIS:

```
(defmethod get-midi  ((s sc-set))
```

28.5.570   sc-set/get-non-chromatic

[sc-set] [Methods]

DESCRIPTION:

Return those notes of a given sc-set object that are microtones (i.e. no
"normal" chromatic notes).

If a number is given for the <octave> argument, the method will transpose
all returned pitches into the specified octave, in which case any duplicate
pitches are removed.

ARGUMENTS:

- An sc-set object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :octave. NIL or an integer that is the octave designator to which all
  resulting pitches are to be transposed (i.e. the "4" in "C4" etc.)
  Default = NIL.
 :as-symbols. T or NIL to indicate whether to return the results of the
  method as a list of note-name symbols rather than a list of pitch
  objects. T = return as note-name symbols. Default = NIL.
- :package. The package in which the pitches are to be handled.
  Default = :sc.

RETURN VALUE:

Returns a list of pitch objects by default.

When the :as-symbols argument is set to T, a list of note-name symbols is
returned instead.

EXAMPLE:

```
;; Returns a list of pitch objects by default
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-non-chromatic mscs))

=>
=> (
PITCH: frequency: 134.646, midi-note: 48, midi-channel: 0
```

```
       pitch-bend: 0.5
       degree: 97, data-consistent: T, white-note: C3
       nearest-chromatic: C3
       src: 0.5146511197090149, src-ref-pitch: C4, score-note: CS3
       qtr-sharp: 1, qtr-flat: NIL, qtr-tone: 1,
       micro-tone: T,
       sharp: NIL, flat: NIL, natural: NIL,
       octave: 3, c5ths: 0, no-8ve: CQS, no-8ve-no-acc: C
       show-accidental: T, white-degree: 21,
       accidental: QS,
       accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: CQS3, tag: NIL,
data: CQS3

PITCH: frequency: 190.418, midi-note: 54, midi-channel: 0
[...]
)

;; Setting :as-symbols to T returns a list of note-names instead
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 d6))))
  (get-non-chromatic mscs
                     :as-symbols t))

=> (CQS3 GQF3 FQS4 BQF5)

;; Giving an integer as the :octave argument transposes all returned  pitches
;; to the specified octave, removing any duplicates
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 e4 fqs4 c5 af5 bqf5 cqs6 d6))))
  (get-non-chromatic mscs
                     :as-symbols t
                     :octave 4))

=> (GQF4 FQS4 BQF4 CQS4)
```

SYNOPSIS:

```
(defmethod get-non-chromatic ((s sc-set)
                              &key
                              (octave nil)
                              (as-symbols nil)
                              (package :sc))
```

28.5.571   sc-set/get-semitones

[sc-set] [Methods]

DESCRIPTION:

 Get the distances in semitones of each pitch in a given sc-set
 object to a static reference pitch.

 Though this method can be used in other contexts, it was devised
 as an aid for transposing audio samples (sound files), and the
 reference pitch is therefore generally the perceived fundamental
 pitch of the audio sample to be transposed.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

 - An optional note-name symbol sets the value of the <reference-pitch>,
   which is the basis pitch to which the resulting number of semitones
   refer. This will generally be the perceived fundamental pitch of the
   sample (sound file) being modified ("transposed").
 - The optional <offset> argument takes a number that is the number of
   semitones to add to the pitch of the given set prior to determining its
   distance in semitones from the reference pitch.

RETURN VALUE:

 A list of positive and negative numbers.

EXAMPLE:

```
;; Chromatic example
(let ((mscs (make-sc-set '(d2 fs3 cs4 c5 af5 d6))))
  (get-semitones mscs))

=> (-22.0 -6.0 1.0 12.0 20.0 26.0)

;; Quarter-tone example; results can be decimal fractions of semitone
(let ((mscs (make-sc-set '(d2 cqs3 fs3 gqf3 cs4 fqs4 c5 af5 bqf5 cqs6 d6))))
  (get-semitones mscs))

=> (-22.0 -11.5 -6.0 -5.5 1.0 5.5 12.0 20.0 22.5 24.5 26.0)
```

SYNOPSIS:

```
(defmethod get-semitones ((s sc-set) &optional
                          (reference-pitch 'c4)
                          (offset 0))
```

28.5.572   sc-set/get-semitones-from-middle-note

[sc-set] [Methods]

DESCRIPTION:

 Return a list of numbers that are the distances in semitones of each pitch
 in a given sc-set object from the middle note of that object.

 NB: If the given sc-object contains an even number of pitch objects, the
     middle note is determined to be the first note of the second half of
     the set.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

 - A symbol that is the key of one of the key/data pairs contained in the
   SUBSETS slot of the given sc-set object.

RETURN VALUE:

 A list of positive and negative numbers.

EXAMPLE:

```
;; With an odd number of items in the sc-set object, the method returns the
;; same number of positive and negative numbers (non-zero)
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5))))
  (get-semitones-from-middle-note mscs))

=> (-21.0 -18.0 -14.0 -11.0 -7.0 -4.0 0.0 3.0 7.0 11.0 14.0 18.0 21.0)

;; With an even number of items in the sc-set object, the middle note is
;; considered to be the first note of the second half of the set
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (get-semitones-from-middle-note mscs))
```

```
=> (-24.0 -21.0 -17.0 -14.0 -10.0 -7.0 -3.0 0.0 4.0 8.0 11.0 15.0 18.0 22.0)

;; Setting the optional <subset> argument to a symbol that is the key of a
;; given key/data pair in the sc-object's SUBSETS slot applies the method to
;; that subset only
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                    :subsets '((fl (df5 f5 af5 c6))
                                 (va (c3 e3 g3 b3 d4 gf4))))))
  (get-semitones-from-middle-note mscs 'fl))

=> (-7.0 -3.0 0.0 4.0)
```

SYNOPSIS:

```
(defmethod get-semitones-from-middle-note ((s sc-set) &optional subset)
```

28.5.573   sc-set/get-srts

[sc-set] [Methods]

DESCRIPTION:

 Get the sampling-rate conversion factors for the given sc-set object,
 whereby 1.0 = unison, 2.0 = one octave higher and 0.5 = one octave lower
 etc.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

 - An optional note-name symbol sets the value of the <reference-pitch>,
   which is the basis pitch to which the resulting factors refer. This will
   generally be the perceived fundamental pitch of the sample (sound file)
   being modified ("transposed").
 - The optional <offset> argument takes a number that is the number of
   semitones to add to the pitch of the given set prior to determining the
   sampling-rate conversion factors.

RETURN VALUE:

 Returns a list of numbers.

EXAMPLE:

```
;; Returns a list of factors that are the sampling-rate conversion factor
;; compared to a 'C4 by default:
(let ((mscs (make-sc-set '(d2 fs3 cs4 c5 af5 d6))))
  (get-srts mscs))

=> (0.28061550855636597 0.7071067690849304 1.0594631433486938 2.0
    3.17480206489563 4.4898481369018555)

;; Comparing the same set against a higher reference-pitch will return lower
;; values
(let ((mscs (make-sc-set '(d2 fs3 cs4 c5 af5 d6))))
  (get-srts mscs 'd4))

=> (0.25 0.6299605220704482 0.9438743681693953 1.781797458637491
    2.8284271254540463 4.0)

;; Conversely, comparing the same set against the default reference-pitch but
;; with a positive offset will return higher values

(let ((mscs (make-sc-set '(d2 fs3 cs4 c5 af5 d6))))
  (get-srts mscs 'c4 2))

=> (0.3149802585215549 0.7937005124004939 1.1892071699914617 2.244924096618746
    3.563594828739576 5.039684136344879)
```

SYNOPSIS:

```
(defmethod get-srts ((s sc-set) &optional
                                (reference-pitch 'c4)
                                (offset 0))
```

28.5.574   sc-set/has-pitches-in-range

[sc-set] [Methods]

DATE:

 June 28th 2017, Edinburgh

DESCRIPTION:

 Tests whether a set has pitches within a certain range.

ARGUMENTS:

```
- the sc-set object
- the lower pitch of the range (pitch object or symbol)
- the upper pitch of the range (pitch object or symbol)
```

OPTIONAL ARGUMENTS:

```
- T or NIL to indicate that only chromatic pitches count, i.e. not
  microtones
```

RETURN VALUE:

```
 T or NIL
```

SYNOPSIS:

```
(defmethod has-pitches-in-range ((s sc-set) lower upper &optional chromatic)
```

28.5.575   sc-set/least-used-octave

[sc-set] [Methods]

DATE:

```
 18th August 2015, Edinburgh
```

DESCRIPTION:

```
 Return the octave with the least notes in it. This includes octaves between
 the lowest and highest notes inclusive, and may include an octave within
 those ranges with no notes at all.
```

ARGUMENTS:

```
 - the sc-set object
```

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :highest-wins. If two octaves share the least number of notes, return the
    highest when T. Default = T.
 - :avoiding. An integer or list of integers. Don't return any octave in
    this list or don't return this single integer argument. Default = NIL.
```

RETURN VALUE:

 An integer representing the octave with the least notes.

EXAMPLE:

```
(least-used-octave (make-sc-set '(cs2 d4 e4 g5)))
--> 3
(least-used-octave (make-sc-set '(cs2 d3 ds4 e4 g5)) :highest-wins nil)
--> 2
(least-used-octave (make-sc-set '(cs2 d3 ds4 e4 g5)))
--> 5
```

SYNOPSIS:

```
(defmethod least-used-octave ((s sc-set) &key (highest-wins t) avoiding)
```

28.5.576   sc-set/make-sc-set

[sc-set] [Functions]

DESCRIPTION:

 Create an sc-set object, which holds pitch-set information for harmonic and
 pitch manipulation.

ARGUMENTS:

 - A list of note-name symbols that is to be the set (pitch-set) for the
   given sc-set object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :id. A symbol that is to be the ID of the given sc-set object.
 - :subsets. An assoc-list of key/data pairs, in which the data is a list of
   note-name symbols that are a subset of the main set. One use for this
   keyword argument might be to create subsets that particular instruments
   can play; these could for instance be selected in the chord-function
   passed to the instrument object. In any case, if the instrument has a
   subset-id slot, and the current set contains a subset with that ID, the
   pitches the instrument may play are limited to that subset.
 - :related-sets. An assoc-list of key/data pairs, similar to :subsets, only
   that the pitches given here do not have to be part of the main set. This
   can be used, for example, for pitches missing from the main set.
 - :auto-sort. T or NIL to indicate whether the specified pitches (note-name
   symbols) are to be automatically sorted from lowest to highest.
   T = sort. Default = T.

RETURN VALUE:

 An sc-set object.

EXAMPLE:

```
;; Simplest usage, with no keyword arguments; returns an sc-set object
(make-sc-set '(d2 cs3 fs3 cs4 e4 c5 af5 ef6))

=>
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 0
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 0, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: USED-NOTES, tag: NIL,
data: NIL

N.B. All pitches printed as symbols only, internally they are all
pitch-objects.

    subsets:
    related-sets:
SCLIST: sclist-length: 8, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (D2 CS3 FS3 CS4 E4 C5 AF5 EF6)

;; With keyword arguments
(make-sc-set '(d2 cs3 fs3 cs4 e4 c5 af5 ef6)
             :id 'scs1
             :subsets '((violin (e4 c5 af5 ef6))
                        (viola (cs4 e4)))
             :related-sets '((missing (ds2 e2 b3 cs6 d6))))

=>
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 0
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
```

```
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 0, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: USED-NOTES, tag: NIL,
data: NIL

N.B. All pitches printed as symbols only, internally they are all
pitch-objects.

    subsets:
VIOLIN: (E4 C5 AF5 EF6)
VIOLA: (CS4 E4)
    related-sets:
MISSING: (DS2 E2 B3 CS6 D6)
SCLIST: sclist-length: 8, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: SCS1, tag: NIL,
data: (D2 CS3 FS3 CS4 E4 C5 AF5 EF6)
```

SYNOPSIS:

```
(defun make-sc-set (sc-set &key id subsets related-sets (auto-sort t) tag
                              (warn-dups t) (rm-dups t)))
```

28.5.577   sc-set/most-used-octave

[sc-set] [Methods]

DATE:

```
 18th August 2015, Edinburgh
```

DESCRIPTION:

```
 Return the octave with the least notes in it. See least-used-octave for
 more details and argument descriptions.
```

SYNOPSIS:

```
(defmethod most-used-octave ((s sc-set) &key (highest-wins t) avoiding)
```

28.5.578   sc-set/pitch-symbols

[sc-set] [Methods]

DESCRIPTION:

Return the pitches of a given sc-set object as a list of note-name
symbols.

ARGUMENTS:

 - An sc-set object.

RETURN VALUE:

 A list of note-name symbols.

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 c3 d4 df5 c6))))
  (pitch-symbols mscs))

=> (D2 C3 D4 DF5 C6)
```

SYNOPSIS:

```
(defmethod pitch-symbols ((s sc-set))
```

28.5.579   sc-set/round-inflections

[sc-set] [Methods]

DESCRIPTION:

 Get the microtones of a given sc-set object, rounded to the nearest
 chromatic note.

 This method returns only the rounded microtones, and not any of the pitches
 of the original sc-set that are already chromatic.

 By default, this method only gets those microtones that are less than a
 quarter-tone. This behavior can be changed by setting the :qtr-tones-also
 argument to T.

 An optional argument allows for all pitches to be moved to a specified
 octave, in which case any duplicate pitches are removed.

ARGUMENTS:

 - An sc-set object.

OPTIONAL ARGUMENTS:

```
 keyword arguments:
 - :qtr-tones-also. T or NIL to indicate whether quarter-tones are also to
   be rounded to the nearest chromatic pitch and returned. T = round and
   return. Default = NIL.
 - :octave. NIL or an integer that is the octave designator to which all
   resulting pitches are to be transposed (i.e. the "4" in "C4" etc.)
   Default = NIL.
 - :remove-duplicates. T or NIL to indicate whether any duplicate pitches
   within an octave that are created by use of the :octave keyword argument
   are to be removed. T = remove duplicates. Default = NIL.
 - :as-symbols. T or NIL to indicate whether to return the results of the
   method as a list of note-name symbols rather than a list of pitch
   objects. T = return as note-name symbols. Default = NIL.
 - :package. The package in which the pitches are to be handled.
   Default = :sc.
```

RETURN VALUE:

```
 A list of pitch objects.
```

EXAMPLE:

```
;; First set the *scale* environment of CM (which is used by slippery chicken)
;; to twelfth-tones
(setf cm::*scale* (cm::find-object 'twelfth-tone))

=> #<tuning "twelfth-tone">

;; By default the method returns a list of pitch objects.
(let ((mscs (make-sc-set '(c4 cts4 css4 cqs4 cssf4 cstf4 cs4))))
  (round-inflections mscs))

=>
(
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
PITCH: frequency: 261.626, midi-note: 60, midi-channel: 0
[...]
data: C4
[...]
PITCH: frequency: 277.183, midi-note: 61, midi-channel: 0
[...]
```

```
data: CS4
[...]
PITCH: frequency: 277.183, midi-note: 61, midi-channel: 0
[...]
data: CS4
)
```

```
;; Setting the :as-symbols argument to T returns a list of note-name symbols
;; instead
(let ((mscs (make-sc-set '(c4 cts4 css4 cqs4 cssf4 cstf4 cs4))))
  (round-inflections mscs :as-symbols t))
```

```
=> (C4 C4 CS4 CS4)
```

```
;; Setting the :qtr-tones-also argument to T returns causes quarter-tones to be
;; rounded and returned as well.
(let ((mscs (make-sc-set '(c4 cts4 css4 cqs4 cssf4 cstf4 cs4))))
  (round-inflections mscs
                     :qtr-tones-also T
                     :as-symbols t))
```

```
=> (C4 C4 C4 CS4 CS4)
```

```
;; Specifying an octave transposes all returned pitches to that octave,
;; removing any duplicates by default
(let ((mscs (make-sc-set '(c2 cts3 css4 cqs5 cssf6 cstf7 cs8))))
  (round-inflections mscs
                     :qtr-tones-also T
                     :octave 4
                     :as-symbols t))
```

```
=> (C4 CS4)
```

```
;; The removal of the duplicates can be turned off by setting the
;; :remove-duplicates argument to NIL
(let ((mscs (make-sc-set '(c2 cts3 css4 cqs5 cssf6 cstf7 cs8))))
  (round-inflections mscs
                     :qtr-tones-also T
                     :octave 4
                     :remove-duplicates NIL
                     :as-symbols t))
```

```
=> (C4 C4 C4 CS4 CS4)
```

SYNOPSIS:

```
(defmethod round-inflections ((s sc-set)
                              &key
                              qtr-tones-also
                              octave
                              (remove-duplicates t) ;; only if octave!
                              (as-symbols nil)
                              (package :sc))
```

28.5.580   sc-set/round-to-nearest

[sc-set] [Methods]

DATE:

 January 30th 2017

DESCRIPTION:

 Round the (potentially very microtonal) pitches in the set to the nearest
 pitch of the current or given scale. Note that this is a destructive
 operation.

ARGUMENTS:

 - the sc-set object

OPTIONAL ARGUMENTS:

 keyword argument:
 :scale. The scale to use when rounding. (Common Music tuning object or
 symbol). If a symbol, then 'chromatic-scale, 'twelfth-tone, or 'quarter-tone
 only at present. Default is the current scale as set by (in-scale :...).

RETURN VALUE:

 the modified sc-set object

SYNOPSIS:

```
(defmethod round-to-nearest ((s sc-set) &key (scale cm::*scale*))
```

28.5.581   sc-set/set-position

[sc-set] [Methods]

DESCRIPTION:

Get the position (zero-index) of a specified pitch object within a given
sc-set object.

ARGUMENTS:

 - A pitch object.
 - An sc-set object.

RETURN VALUE:

 An integer.

EXAMPLE:

```
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (set-position (make-pitch 'e3) mscs))

=> 4
```

SYNOPSIS:

```
(defmethod set-position ((p pitch) (s sc-set))
```

28.5.582   sc-set/stack

[sc-set] [Methods]

DESCRIPTION:

 Extend the pitch content of a given sc-set object by adding new pitch
 objects which have the same interval structure as the original set.

 The method analyzes the interval structure of the original set from the
 bottom note to the top and adds new sets to the top and bottom of the
 original set symmetrically; i.e., with the identical interval structure
 above the original set and inverted interval structure below.

 The second argument indicates how times this procedure is carried out.

 NB: The method assumes that the pitch content of the original sc-set object
     is sorted from low to high.

 See also: the make-stack method in the complete-set class to make a stack
           from a simple list of note-name symbols.

ARGUMENTS:

- An sc-set object.
- An integer that is the number of new sets to be added to each end of the
  original set.

OPTIONAL ARGUMENTS:

keyword arguments:
- :id. A symbol that will be the ID of the new sc-set object. Default NIL.
- :by-freq. If T then use frequencies when calculating the interval
  structure instead of degrees (semitones if default scale is chromatic).
  In this case the frequencies of pitches will be retained but their
  symbolic value will be rounded to the nearest note in the current scale
  (and pitch bends will be set accordingly). Default NIL.
- :up. Apply the process upwards in pitch space. Default = T.
- :down. Apply the process downwards in pitch space. Default = T.

RETURN VALUE:

A new sc-set object. This will have the same tag and id (unless given as a
keyword argument here) slots as the original sc-set but not subsets,
related-sets, used-notes, etc.

EXAMPLE:

```
;; Extends the original set with new sets that have the identical interval
;; structure upwards and inverted interval structure downwards.
(let ((set (make-sc-set '(c4 e4 g4))))
  (stack set 3))
=>
SC-SET: auto-sort: T, used-notes:
[...]
data: (EF2 GF2 BF2 DF3 F3 AF3 C4 E4 G4 B4 D5 GF5 A5 DF6 E6)

;;; or by calling the make-stack function, which returns a complete-set object
;;; (subclass of tl-set and sc-set).  Called with (in-scale :chromatic):
(make-stack 'test '(430 441 889 270) 1 :by-freq t)
=>
COMPLETE-SET: complete: NIL
[...]
data: (G2 A2 CS4 A4 A4 A5 C6 C6 FS6)
```

SYNOPSIS:

```
(defmethod stack ((s sc-set) num-stacks &key id by-freq (up t) (down t))
```

28.5.583   sc-set/subset-get-srts

[sc-set] [Methods]

DESCRIPTION:

Get the sampling-rate conversion factors for the specified subset of a
given sc-set object, whereby 1.0 = unison, 2.0 = one octave higher and 0.5
= one octave lower etc.

ARGUMENTS:

- An sc-set object.
- A symbol that is the key of one of the key/data pairs stored in the
  SUBSETS slot of the given sc-set object.

OPTIONAL ARGUMENTS:

- The optional <reference-pitch> is the basis pitch to which the resulting
  factors refer. This will generally be the perceived fundamental pitch of
  the sample (sound file) being modified ("transposed").
- The optional <offset> argument is the number of semitones to add to the
  pitch of the given set prior to determining the sampling-rate conversion
  factors.

RETURN VALUE: EXAMPLE:

```
;;; Create an sc-set object with two subsets named 'FL and 'VA, then get the
;;; sampling-rate conversion factors for the 'FL subset
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                    :subsets '((fl (df5 f5 af5 c6))
                               (va (c3 e3 g3 b3 d4 gf4))))))
  (subset-get-srts mscs 'fl))

=> (2.1189262866973877 2.669679641723633 3.17480206489563 4.0)
```

SYNOPSIS:

```
(defmethod subset-get-srts  ((s sc-set) subset &optional
                             (reference-pitch 'c4)
                             (offset 0))
```

28.5.584   sc-set/subsets-remove

[sc-set] [Methods]

DATE:

June 2nd 2017, Edinburgh

DESCRIPTION:

 Remove pitches from all subsets of an sc-set object

ARGUMENTS:

 - the sc-set object
 - &rest the pitches to remove, as pitch objects or symbols

RETURN VALUE:

 the sc-set

SYNOPSIS:

(defmethod subsets-remove ((s sc-set) &rest pitches)

28.5.585   sc-set/thin

[sc-set] [Methods]

DATE:

 June 1st 2017, Edinburgh

DESCRIPTION:

 Remove pitches from a set using a deterministic algorithm. From bottom to
 top (or vice-versa) pitches will be selected or rejected using the
 activity-levels class, to which the strength (1-10) is passed. The pitches
 removed are also removed from any subsets.

 By default the set will be reduced by 1/3 of its pitches but it is expected
 that either the :remove or :target keyword will be used (see below).

ARGUMENTS:

 - an sc-set object

OPTIONAL ARGUMENTS:

keyword arguments:
- :strength. An integer from 1-10 indicating the likelihood of removing
  individual pitches. 1 would be 10%, 10, 100%. Default = 5.
- :remove. An integer indicating how many pitches to remove. Default = NIL
- :target. An integer indicating how many pitches the set should be left
  with. Default = NIL
- :invert. T or NIL to remove pitches starting at the top or bottom. T =
  from top, NIL = from bottom. Default = NIL

RETURN VALUE:

 The (thinned) sc-set object.

EXAMPLE: SYNOPSIS:

(defmethod thin ((s sc-set) &key (strength 5) remove target invert)

28.5.586 sc-set/tl-set

[sc-set] [Classes]

NAME:

 tl-set

 File: tl-set.lsp

 Class Hierarchy: named-object -> linked-named-object -> sclist -> chord ->
                  sc-set -> tl-set

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the tl-set class that extends
                  set to incorporate transposition and limiting to
                  certain pitch ranges. NB As of yet, once a set is
                  transposed or limited, it can't be retransposed from its
                  original pitches, only from the current set, i.e these
                  methods are destructive!

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: 13th August 2001

```
$$ Last modified:  09:57:02 Fri Jun 21 2019 CEST
```

```
SVN ID: $Id$
```

28.5.587   tl-set/complete-set

[tl-set] [Classes]

NAME:

```
complete-set
```

```
File:             complete-set.lsp
```

```
Class Hierarchy:  named-object -> linked-named-object -> sclist -> chord ->
                  sc-set -> tl-set -> complete-set
```

```
Version:          1.0.11
```

```
Project:          slippery chicken (algorithmic composition)
```

```
Purpose:          Implementation of the complete-set class which as an
                  extension of the tl-set class allows checking for full
                  sets: ones in which every note of the current scale is
                  present.
```

```
Author:           Michael Edwards: m@michael-edwards.org
```

```
Creation date:    10th August 2001
```

```
$$ Last modified:  12:41:50 Sat May  2 2020 CEST
```

```
SVN ID: $Id$
```

28.5.588   complete-set/make-complete-set

[complete-set] [Functions]

DESCRIPTION:

```
Create a complete-set object, which as an extension of the tl-set class
allows checking for full sets: ones in which every note of *standard-scale*
is present.
```

ARGUMENTS:

- A set of pitches. This can either take the form of a list of note-name
  symbols or a complete-set, tl-set or sc-set object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :id. An number, symbol or string that is to be the ID of the given
  complete-set object (see doc for sc-set).
- :tag. A number, symbol or string that is secondary name, description, tag
  etc. for the given complete-set object. The :tag serves for
  identification but not searching purposes (see doc for named-object).
- :subsets. An assoc-list of key/data pairs, in which the data is a list of
  note-name symbols that are a subset of the main set. One use for this
  keyword argument might be to create subsets that particular instruments
  can play; these would then be selected in the chord-function passed to
  the instrument object (see doc for sc-set).
- :related-sets. An assoc-list of key/data pairs, similar to :subsets, only
  that the pitches given here do not have to be part of the main set. This
  can be used, for example, for pitches missing from the main set (see doc
  for sc-set).
- :auto-sort. T or NIL to indicate whether the specified pitches (note-name
  symbols) are to be automatically sorted from lowest to highest.
  T = sort. Default = T. (see doc for sc-set)
- :transposition. A number that is the number of semitones by which the
  pitches of the new complete-set are to be transposed when the object is
  created. Default = 0. (see doc for tl-set)
- :limit-upper. A note-name symbol or a pitch object to indicate the
  highest possible pitch in the given complete-set object to be
  created. (see doc for tl-set)
- :limit-lower. A note-name symbol or a pitch object to indicate the
  lowest possible pitch in the complete-set object to be created. (see doc
  for tl-set)
- :complete. T, NIL, or 'CHROMATIC. This argument can be given at init, and
  if the set is not complete in the sense of T or 'CHROMATIC (all
  chromatic, equally-tempered notes are present in the set), a warning is
  printed. If the set is neither T nor 'CHROMATIC at init, then no warning
  will be issued. In both cases the COMPLETE slot of the given complete-set
  object will be set after checking the set.
- :midi-channel. The channel to set chromatic pitches to. Default = 1.
- :microtones-midi-channel The channel to set microtonal pitches to.
  Default = 2.

RETURN VALUE:

A complete-set object.

EXAMPLE:

```
;; Create a complete set using a list of note-name symbols and the default
;; values for the keyword arguments
(make-complete-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))

=>
COMPLETE-SET: complete: NIL
...
data: (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5 C6)

;; A new complete-set object can be created from tl-set and sc-set objects
(let ((mcs (make-complete-set
            (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5)))))
  (pitch-symbols mcs))

=> (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5)

(let ((mcs (make-complete-set
            (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5)))))
  (pitch-symbols mcs))

=> (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5)

;; Using the other keyword arguments
(make-complete-set '(d2 f2 a2 e3 g3 b3 d4 gf4 bf4 df5 f5 af5)
            :id 'csset
            :subsets '((low (d2 f2 a2))
                       (mid (b3 d4)))
            :related-sets '((not-playable (dqs2 eqf3)))
            :transposition 3
            :limit-upper 'g5
            :limit-lower 'e2)

=>
COMPLETE-SET: complete: NIL
...
data: (F2 AF2 C3 G3 BF3 D4 F4 A4 CS5 E5)
```

SYNOPSIS:

```
(defun make-complete-set (set &rest keys
                         &key id tag subsets related-sets
                           (transposition 0) (auto-sort t) (warn-dups t)
                           (rm-dups t) limit-upper limit-lower complete
                           ;; MDE Fri Aug 24 14:56:50 2018
```

```
                            (midi-channel 1) (microtones-midi-channel 2))
```

28.5.589   complete-set/make-stack

[complete-set] [Functions]

DESCRIPTION:

Make a complete-set containing stacks based on the given notes. See
documentation for the stack method in the sc-set class for details of
what this algorithm does.

ARGUMENTS:

- an ID for the set to be created (symbol, string, number)
- a list of notes as either symbols or pitch objects
- the number of stacks to create

OPTIONAL ARGUMENTS:

keyword arguments:
- :by-freq. Use the frequencies of the pitches to create the stack instead
  of the interval structure. Default = NIL.
- :up. Apply the process upwards in pitch space. Default = T.
- :down. Apply the process downwards in pitch space. Default = T.

RETURN VALUE:

a complete-set object

SYNOPSIS:

(defun make-stack (id notes num-stacks &key by-freq (up t) (down t))

28.5.590   tl-set/limit

[tl-set] [Methods]

DESCRIPTION:

Remove pitch objects from a given tl-set whose pitch content is higher or
lower than the pitches specified. Any pitch objects whose pitch content is
equal to the limit pitches specified will be retained.

NB: C0 and B10 are the highest and lowest possible pitches of the
    quarter-tone scale defined in scale.lsp (16.35 and 31608.55 Hz
    respectively).

NB: The keyword arguments for which the lower and upper limits are to be
    specified are optional arguments, but are required in order for any
    effect to be had.

NB: Whether a pitch is higher or lower than the specified pitches is
    determined by their frequencies. If a pitch happens to be microtonal it
    could be that the nearest pitch symbol is an approximation, i.e. not
    quite the same as the frequency, so limiting might appear to be working
    incorrectly.

ARGUMENTS:

 - The tl-set object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :upper. A note-name symbol that is the upper limit for the limiting
   process.
 - :lower. A note-name symbol that is the lower limit for the limiting
   process.
 - :do-related-sets. T or NIL to indicate whether the RELATED-SETS slot of
   the given tl-set object is to be transposed as well or left unhandled. T
   = transpose. Default = NIL.

RETURN VALUE:

 A tl-set object.

EXAMPLE:

```
;;; By default the method does not transpose the pitches of the RELATED-SETS
;;; slot
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                    :subsets '((fl (df5 f5 af5))
                               (vla (e3 g3 b3)))
                    :related-sets '((missing (fs2 b5))))))
  (limit mtls :upper 'df5 :lower 'c3))

=>
TL-SET: transposition: 0
```

```
        limit-upper:
PITCH: frequency: 554.365, midi-note: 73, midi-channel: 0
[...]
    subsets:
FL: (DF5)
VLA: (E3 G3 B3)
    related-sets:
MISSING: (FS2 B5)
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (C3 E3 G3 B3 D4 GF4 BF4 DF5)


;; Setting the :do-related-sets argument to T results in any RELATED-SETS pitch
;; content being transposed as well
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                         :subsets '((fl (df5 f5 af5))
                                    (vla (e3 g3 b3)))
                         :related-sets '((missing (fs2 b5))))))
  (limit mtls :upper 'c6 :lower 'c3 :do-related-sets t))

=>
[...]
    subsets:
FL: (DF5 F5 AF5)
VLA: (E3 G3 B3)
    related-sets:
MISSING: (B5)
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5 C6)
```

SYNOPSIS:

```
(defmethod limit ((tls tl-set) &key upper lower do-related-sets)
```

28.5.591   tl-set/limit-for-instrument

[tl-set] [Methods]

DESCRIPTION:

 Remove any pitch objects from the given tl-set object which are outside of
 the range of the specified instrument object.

The pitch objects returned after that operation can then be reduced again
by applying further limits specified by the :upper and :lower keyword
arguments.

NB: This method returns a list of pitch objects, not a tl-set object,
    though it does destructively alter the data of the given tl-set object
    accordingly.

NB: This method will return NIL if the pitch objects of the given tl-set
    object are microtonal while the given instrument object is set to be a
    non-microtonal instrument (see example).

NB: Whether a pitch is higher or lower than the specified pitches is
    determined by their frequencies. If a pitch happens to be microtonal it
    could be that the nearest pitch symbol is an approximation, i.e. not
    quite the same as the frequency, so limiting might appear to be working
    incorrectly.

ARGUMENTS:

 - A tl-set object.
 - An instrument object.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :upper. A pitch object or note-name symbol that is the uppermost possible
   pitch (inclusive) of the pitch objects returned, as a further limitation
   after the range of the instrument object has been applied.
 - :lower. A pitch object or note-name symbol that is the lowermost possible
   pitch (inclusive) of the pitch objects returned, as a further limitation
   after the range of the instrument object has been applied.
 - :do-related-sets. T or NIL to indicate whether to apply the specified
   range restrictions to the RELATED-SETS slot of the given tl-set object as
   well. NB: These will be modified within the original tl-set object but
   not returned as part of the resulting list. T = apply to RELATED-SETS as
   well. Default = NIL.

RETURN VALUE:

 A list of pitch objects.

EXAMPLE:

```
;;; Returns a list of pitch objects, limited only by the range of the given
;;; instrument object by default
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                          :related-sets '((other-notes (b4 e5 fs5 c6)))))
      (mi (make-instrument 'flute
                            :staff-name "Flute" :staff-short-name "Fl."
                            :lowest-written 'c4 :highest-written 'd7
                            :starting-clef 'treble :midi-program 74 :chords nil
                            :microtones t :missing-notes '(cqs4 dqf4))))
  (limit-for-instrument mtls mi))

=>
(
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 0
[...]
data: D4
[...]
PITCH: frequency: 369.994, midi-note: 66, midi-channel: 0
[...]
data: GF4
[...]
PITCH: frequency: 466.164, midi-note: 70, midi-channel: 0
[...]
data: BF4
[...]
PITCH: frequency: 554.365, midi-note: 73, midi-channel: 0
[...]
data: DF5
[...]
PITCH: frequency: 698.456, midi-note: 77, midi-channel: 0
[...]
data: F5
[...]
PITCH: frequency: 830.609, midi-note: 80, midi-channel: 0
[...]
data: AF5
[...]
PITCH: frequency: 1046.502, midi-note: 84, midi-channel: 0
[...]
data: C6
)

;;; Further restrict the pitches returned by setting values for the :upper and
;;; :lower keyword arguments and print the new pitch content of the given
;;; tl-set object to see the destructive modification
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
```

```
                                :related-sets '((other-notes (b4 e5 fs5 c6)))))
      (mi (make-instrument 'flute
                              :staff-name "Flute" :staff-short-name "Fl."
                              :lowest-written 'c4 :highest-written 'd7
                              :starting-clef 'treble :midi-program 74 :chords nil
                              :microtones t :missing-notes '(cqs4 dqf4))))
  (limit-for-instrument mtls mi :upper 'b5 :lower 'c5)
  (pitch-symbols mtls))

=> (DF5 F5 AF5)


;;; By default the RELATED-SETS slot of the given tl-set object is not affected
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                            :related-sets '((other-notes (b4 e5 fs5 c6)))))
      (mi (make-instrument 'flute
                              :staff-name "Flute" :staff-short-name "Fl."
                              :lowest-written 'c4 :highest-written 'd7
                              :starting-clef 'treble :midi-program 74 :chords nil
                              :microtones t :missing-notes '(cqs4 dqf4))))
  (limit-for-instrument mtls mi :upper 'b5 :lower 'c5)
  (loop for nobj in (data (related-sets mtls))
     collect (loop for p in (data nobj)
                collect (data p))))

=> ((B4 E5 FS5 C6))


;;; Setting the :do-related-sets argument to T will cause the method to be
;;; applied to the RELATED-SETS slot as well.
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                            :related-sets '((other-notes (b4 e5 fs5 c6)))))
      (mi (make-instrument 'flute
                              :staff-name "Flute" :staff-short-name "Fl."
                              :lowest-written 'c4 :highest-written 'd7
                              :starting-clef 'treble :midi-program 74 :chords nil
                              :microtones t :missing-notes '(cqs4 dqf4))))
  (limit-for-instrument mtls mi :upper 'b5 :lower 'c5 :do-related-sets t)
  (print (pitch-symbols mtls))
  (print (loop for nobj in (data (related-sets mtls))
            collect (loop for p in (data nobj)
                       collect (data p)))))

=>
(DF5 F5 AF5)
((E5 FS5))
```

```
;;; The method will return NIL if a set of only microtonal pitches (which
;;; e.g. ring-mod might return) is given in combination with an instrument
;;; object which is not microtone-capable (such as the 'piano object of the
;;; +slippery-chicken-standard-instrument-palette+
(let ((mtls (make-tl-set '(dqs2 fqs2 aqf2 gqs3 bqf3 gqf4 bqf4 dqf5 fqs5)))
      (pno (get-data 'piano
                     +slippery-chicken-standard-instrument-palette+)))
  (limit-for-instrument mtls pno :lower 'e5 :upper 'd6))

=> NIL
```

SYNOPSIS:

```
(defmethod limit-for-instrument ((tls tl-set) (ins instrument)
                                 &key upper lower do-related-sets)
```

28.5.592   tl-set/limit-shift-octave

[tl-set] [Methods]

DATE:

August 18th 2015, Edinburgh

DESCRIPTION:

Like the limit method, we restrict pitches in the set to be within
specified limits, but instead of removing pitches we shift them to octaves
within the limits. Rather than just putting the outlying pitches up or down
and octave or two, we put them into the least-used octave of the set so as
to balance pitch spread.

ARGUMENTS:

- the tl-set object

OPTIONAL ARGUMENTS:

keyword arguments:
- :upper. A note-name symbol that is the upper limit for the limiting
  process.
- :lower. A note-name symbol that is the lower limit for the limiting
  process.
- :do-related-sets. T or NIL to indicate whether the RELATED-SETS slot of
  the given tl-set object is to be transposed as well or left unhandled. T

```
   = transpose. Default = NIL.
 - :highest-wins. T or NIL to indicate whether in the case of a tie when
   searching for the least used octave in a set, the highest octave will be
   used to move notes into. Default = T.
```

RETURN VALUE:

```
 The tl-set object.
```

EXAMPLE:

```
;;; the C2 will be shifted to C5 as the 5th octave is the least used in the
;;; original set. The DS5 will be shifted down to DS2 as once we've shifted the
;;; C2, the second octave is now the least used, plus it's within range.
(get-pitch-symbols
 (limit-shift-octave (make-tl-set '(c2 e2 cs3 f3 g3 d4 a4 ds5))
                     :upper 'c5 :lower 'd2))
--> (DS2 E2 CS3 F3 G3 D4 A4 C5)
```

SYNOPSIS:

```
(defmethod limit-shift-octave ((tls tl-set)
                               &key upper lower (highest-wins t)
                                 do-related-sets)
```

28.5.593   tl-set/make-tl-set

[tl-set] [Functions]

DESCRIPTION:

```
 Create a tl-set object, which extends the sc-set class by incorporating
 transposition and limiting to certain pitch ranges.

 NB: As of yet, once a set is transposed or limited, it can't be
     re-transposed from its original pitches, only from the current set; i.e
     these methods are destructive!
```

ARGUMENTS:

```
 - A list of note-name symbols that is to be the set (pitch-set) for the
   given tl-set object.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :id. A symbol that is to be the ID of the given tl-set object.
- :subsets. An assoc-list of key/data pairs, in which the data is a list of
  note-name symbols that are a subset of the main set. One use for this
  keyword argument might be to create subsets that particular instruments
  can play; these would then be selected in the chord-function passed to
  the instrument object.
- :related-sets. An assoc-list of key/data pairs, similar to :subsets, only
  that the pitches given here do not have to be part of the main set. This
  can be used, for example, for pitches missing from the main set.
- :limit-upper. A note-name symbol or a pitch object to indicate the
  highest possible pitch in the tl-set object to be created.
- :limit-lower.  A note-name symbol or a pitch object to indicate the
  lowest possible pitch in the tl-set object to be created.
- :transposition. A number that is the number of semitones by which the
  pitches of the new tl-set are to be transposed when the object is
  created. Default = 0.
- :auto-sort. T or NIL to indicate whether the specified pitches (note-name
  symbols) are to be automatically sorted from lowest to highest. T =
  sort. Default = T.
```

RETURN VALUE:

```
 A tl-set object.
```

EXAMPLE:

```
;; Simple usage with default values for keyword arguments
(make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))

=>
TL-SET: transposition: 0
        limit-upper: NIL
        limit-lower: NIL
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
                      num-data: 0
                      linked: NIL
                      full-ref: NIL
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 0, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: USED-NOTES, tag: NIL,
data: NIL
```

N.B. All pitches printed as symbols only, internally they are all
pitch-objects.

```
    subsets:
    related-sets:
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5 C6)

;; Adding subsets and related-sets
(make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
             :subsets '((fl (df5 f5 af5))
                        (vla (e3 g3 b3)))
             :related-sets '((missing (fs2 b5))))
=>
TL-SET: transposition: 0
[...]
    subsets:
FL: (DF5 F5 AF5)
VLA: (E3 G3 B3)
    related-sets:
MISSING: (FS2 B5)
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (D2 F2 A2 C3 E3 G3 B3 D4 GF4 BF4 DF5 F5 AF5 C6)

;; Limiting the pitch range of the tl-set object, once using a note-name
;; symbol and once using a pitch object
(make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
             :limit-upper 'g5
             :limit-lower (make-pitch 'd3))

=>
TL-SET: transposition: 0
        limit-upper:
PITCH: frequency: 783.991, midi-note: 79, midi-channel: 0
[...]
        limit-lower:
PITCH: frequency: 146.832, midi-note: 50, midi-channel: 0
[...]
data: (E3 G3 B3 D4 GF4 BF4 DF5 F5)

;; Applying a transposition by semitones
(make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
```

```
              :transposition 3)

=>
TL-SET: transposition: 3
[...]
data: (F2 AF2 C3 EF3 G3 BF3 D4 F4 A4 CS5 E5 AF5 B5 EF6)
```

SYNOPSIS:

```
(defun make-tl-set (set &key id subsets related-sets
                              limit-upper limit-lower
                              (rm-dups t)
                              (transposition 0)
                              (auto-sort t))
```

28.5.594   tl-set/stack

[tl-set] [Methods]

DESCRIPTION:

 Extend the pitch content of a given sc-set object by adding new pitch
 objects which have the same interval structure as the original set.

 NB: See documentation for the stack method in the sc-set class for usage.

SYNOPSIS:

```
(defmethod stack ((tls tl-set) num-stacks &key id by-freq)
```

28.5.595   tl-set/transpose

[tl-set] [Methods]

DESCRIPTION:

 Transpose the pitches of a given tl-set by a specified number of
 semitones.

 The contents of the SUBSETS slot are automatically transposed as well, but
 the RELATED-SETS slot is left untransposed by default. An optional argument
 allows for RELATED-SETS slot to be transposed as well.

ARGUMENTS:

  - A tl-set object.
  - A positive or negative integer that is the number of semitones by which
    the pitch content of the given tl-set object is to be transposed.

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :do-related-sets. T or NIL to indicate whether to transpose any contents
  of the RELATED-SETS slot as well. T = transpose.  Default = NIL.
(- additional <ignore> arguments are for internal use only)
```

RETURN VALUE:

```
The tl-set object.
```

EXAMPLE:

```
;; By default the RELATED-SETS are left untransposed
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                          :subsets '((fl (df5 f5 af5))
                                     (vla (e3 g3 b3)))
                          :related-sets '((missing (fs2 b5))))))
  (transpose mtls 3))

=>
TL-SET: transposition: 3
        limit-upper: NIL
        limit-lower: NIL
SC-SET: auto-sort: T, used-notes:
[...]
    subsets:
FL: (E5 AF5 B5)
VLA: (G3 BF3 D4)
    related-sets:
MISSING: (FS2 B5)
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
pNAMED-OBJECT: id: NIL, tag: NIL,
data: (F2 AF2 C3 EF3 G3 BF3 D4 F4 A4 CS5 E5 AF5 B5 EF6)

;; Set the <do-related-sets> argument to T for the RELATED-SETS contents to be
;; transposed as well
(let ((mtls (make-tl-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6)
                          :subsets '((fl (df5 f5 af5))
                                     (vla (e3 g3 b3)))
```

```
                              :related-sets '((missing (fs2 b5))))))
  (transpose mtls 3 :do-related-sets t))

=>
TL-SET: transposition: 3
        limit-upper: NIL
        limit-lower: NIL
SC-SET: auto-sort: T, used-notes:
[...]
    subsets:
FL: (E5 AF5 B5)
VLA: (G3 BF3 D4)
    related-sets:
MISSING: (A2 D6)
SCLIST: sclist-length: 14, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (F2 AF2 C3 EF3 G3 BF3 D4 F4 A4 CS5 E5 AF5 B5 EF6)
```

SYNOPSIS:

```
(defmethod transpose :before ((tls tl-set) semitones
                                &key destructively do-related-sets
                                  ;; MDE Wed Aug 22 09:43:29 2018
                                  lowest highest)
```

28.5.596   sclist/sc-subseq

[sclist] [Methods]

DESCRIPTION:

 Return a subsequence from a given sclist based on starting and finishing
 indices.

 NB: This method uses Common Lisp's subseq function and thus inherits its
 attributes, whereby the START argument indicates the zero-based index of
 the first list item to be returned and the FINISH argument indicates the
 zero-based index of the first list item after that NOT to be returned.

ARGUMENTS:

 - An sclist object.
 - An integer indicating the zero-based index of the first list item to be
   returned.

   - An integer indicating the zero-based index of the first list item after
     the START item to not be returned.

OPTIONAL ARGUMENTS:

 - (fun #'error). By default an error will be signalled if the requested
 subseq is out of bounds. If you prefer, this could be a warning instead by
 passing #'warn, or nothing at all if NIL.

RETURN VALUE:

 A list.

 An error is returned if the user attempts to apply the method with START
 and FINISH arguments that are beyond the bounds of the given sclist
 object.

EXAMPLE:

```
;; Returns a sublist from the given list. The START argument indicates the
;; zero-based index of the first item in the given list to be returned and the
;; FINISH argument indicates the zero-based index of the first item after that
;; to NOT be returned.
(let ((scl (make-sclist '(1 2 3 4 5 6 7 8 9))))
  (sc-subseq scl 2 7))

=> (3 4 5 6 7)

;; Drops into the debugger with an error if one of the indexing arguments is
;; beyond the bounds of the given sclist object
(let ((scl (make-sclist '(1 2 3 4 5 6 7 8 9))))
  (sc-subseq scl 0 15))

=>
sclist::sc-subseq: Illegal indices for above list: 0 15 (length = 9)
   [Condition of type SIMPLE-ERROR]

(let ((scl (make-sclist '(1 2 3 4 5 6 7 8 9))))
  (sc-subseq scl 0 15 NIL))
=>
NIL
```

SYNOPSIS:

```
(defmethod sc-subseq ((scl sclist) start finish &optional (fun #'error))
```

28.5.597 sclist/sclist-econs

[sclist] [Methods]

DESCRIPTION:

 Add a single item to the end of a given sclist object.

 NB: This method destructively modifies the list.

 NB: This method adds any element specified as a single item. For combining
     two lists into one see sclist/combine.

 NB: Though related to Lisp's cons function, remember that the order of
     arguments here is the other way round i.e. element after list, not
     before.

ARGUMENTS:

 - An sclist object.
 - An item to add to the end of the given sclist object.

RETURN VALUE:

 - The new value (list) of the given sclist object.

EXAMPLE:

```
;; Add a single integer to the end of a list of integers
(let ((scl (make-sclist '(0 1 2 3 4))))
  (sclist-econs scl 5))

=> (0 1 2 3 4 5)
```

SYNOPSIS:

```
(defmethod sclist-econs ((scl sclist) element)
```

28.5.598 sclist/sclist-remove-elements

[sclist] [Methods]

DESCRIPTION:

 Remove a specified number of consecutive items from a given sclist object.

NB: This is a destructive method and replaces the data of the given sclist
object with the newly created sublist.

ARGUMENTS:

 - An sclist object.
 - The index integer within the given list with which to start (inclusive
   and zero-based).
 - An integer that is the number of items to remove.

RETURN VALUE:

 Returns

EXAMPLE:

```
;;; Returns an sclist object.
(let ((scl (make-sclist '(0 1 2 3 4 5 6 7 8 9))))
  (sclist-remove-elements scl 3 4))

=>
SCLIST: sclist-length: 6, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: (0 1 2 7 8 9)

;; Drops into the debugger with an error if the given sclist object has fewer
;; items than specified for the START or HOW-MANY arguments
(let ((scl (make-sclist '(0 1 2 3 4 5 6 7 8 9))))
  (data (sclist-remove-elements scl 3 41)))

=>
remove-elements: arguments 2 and 3 must be integers < the length of argument 1:
3 41 10
   [Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod sclist-remove-elements ((scl sclist) start how-many)
```


28.5.599   sclist/set-nth

[sclist] [Methods]

DESCRIPTION:

 Set the nth element (zero-based) of a given sclist object.

 NB: This doesn't auto-grow the list.

ARGUMENTS:

 - An index integer.
 - An sclist object.

RETURN VALUE:

 Returns the item added if successful.

 Returns NIL and prints a warning if the specified index number is greater
 than the number of items in the list (minus 1)

EXAMPLE:

```
;; Returns the item added
(let ((scl (make-sclist '(cat dog cow pig sheep))))
  (set-nth 3 'horse scl))

=> HORSE

;; Access the DATA slot to see the change
(let ((scl (make-sclist '(cat dog cow pig sheep))))
  (set-nth 3 'horse scl)
  (data scl))

=> (CAT DOG COW HORSE SHEEP)

;; Returns NIL and prints a warning if the index number is beyond the bounds of
;; the list
(let ((scl (make-sclist '(cat dog cow pig sheep))))
  (set-nth 31 'horse scl))

=> NIL
WARNING: sclist::sclist-check-bounds: Illegal list reference: 31
(length = 5) (sclist id = NIL)
```

SYNOPSIS:

```
(defmethod set-nth (index new-element (i sclist)))
```

28.5.600   linked-named-object/sndfile

[linked-named-object] [Classes]

NAME:

 sndfile

 File: sndfile.lsp

 Class Hierarchy: named-object -> linked-named-object -> sndfile

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the sndfile class that holds
                   information about a sound file as well as specifying
                   desired parameters

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: March 21st 2001

 $$ Last modified: 19:18:35 Thu Sep 24 2020 CEST

 SVN ID: Id

28.5.601   sndfile/make-sndfile

[sndfile] [Functions]

DESCRIPTION:

 Create a sndfile object to hold data about an existing sound file,
 specifying at least the path and file name of that sound file. Optional
 arguments allow for the specification of segments within the given sound
 file and its perceived fundamental frequency (for src-based
 transposition).

 If the first argument ("path") is a list, then this sndfile object has been
 created from within a sndfile-palette object. In this case, the first item
 in the list will be the full path to the sound file, as defined in the
 sndfile-palette object; the second item in the list is the list given by
 the user containing the data slots to be used, whereby the first item of
 that list must be the ID of the object.

NB: This function creates an object of the class sndfile, which is contains
   data concerning an existing sound file. It does not create an actual
   sound file.

ARGUMENTS:

- A path and file name of an existing sound file; or a list as explained
  above.

OPTIONAL ARGUMENTS:

keyword arguments:
- :id. An ID for the sndfile. Will be set automatically if created from
  within a sndfile-palette. Default nil.
- :data. The given file name, including path and extension, usually set
  automatically to be the given path if nil. Default nil.
- :duration. A number in seconds which is the duration of the segment of
  the specified sound file which the user would like to use. This should
  not be specified if :end has been specified. Default nil.
- :end. A number in seconds which is the end time within the source sound
  file for the segment of the file which the user would like to use. This
  should not be specified if :duration has been specified. Default nil.
- :start. A number in seconds which is the start time within the source
  sound file for the segment of the file which the user would like to
  use. Defaults to 0.0.
- :frequency. A number or note-name symbol. This frequency will serve as
  the reference pitch for any src transpositions of this file. This can be
  any value, but will most likely be specified if the source sound file has
  a perceptible fundamental pitch. If given as a number, this number will
  be handled as a frequency in Hertz. Default = 'C4.
- :amplitude. An number that is the amplitude which the user would like to
  designate for this sound file. This number may be of any value, as
  slippery chicken normalizes all sound file events; however, standard
  practice would suggest that this should fall between 0.0 and 1.0.
  Default = 1.0

RETURN VALUE:

A sndfile object.

EXAMPLE:

```
;; Example specifying the full path, a start and end time, and a base frequency
(make-sndfile "/path/to/sndfile-1.aiff"
```

```
                :start 0.3
                :end 1.1
                :frequency 654)

=>


SNDFILE: path: /path/to/sndfile-1.aiff,
        snd-duration: 3.011043, channels: 1, frequency: 654
        start: 0.3, end: 1.1, amplitude: 1.0, duration 0.8
        will-be-used: 0, has-been-used: 0
        data-consistent: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: /path/to/sndfile-1.aiff

;; Example using the sndfile-palette list as the first argument
(make-sndfile '("/path/to/sndfile-1.aiff"
                (nil :start 0.3 :end 1.1)))

=>


SNDFILE: path: /path/to/sndfile-1.aiff,
        snd-duration: 3.011043, channels: 1, frequency: 261.62555
        start: 0.3, end: 1.1, amplitude: 1.0, duration 0.8
        will-be-used: 0, has-been-used: 0
        data-consistent: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "/Volumes/JIMMY/SlipperyChicken/sndfile-2.aiff", tag: NIL,
data: /path/to/sndfile-1.aiff
```

SYNOPSIS:

```
(defun make-sndfile (path &key id data duration end (start 0.0)
                     (frequency nil)
                     (amplitude 1.0))
```

28.5.602   sndfile/reset-usage

[sndfile] [Methods]

DESCRIPTION:

Reset the WILL-BE-USED and HAS-BEEN-USED slots of the given sndfile object
to 0. These slots keep track of how many times a sound will be used and has
been used, which is useful for purposes such as incrementing

start-time. These slots are set internally and are not intended to be set by
the user.

ARGUMENTS:

- A sndfile object.

RETURN VALUE:

Returns 0.

EXAMPLE:

```
;; First set the values of the WILL-BE-USED and HAS-BEEN-USED slots, as these
;; are 0 when a new sndfile object is created using make-sndfile. Set the
;; values, print them; reset both using reset-usage, and print again to see
;; the change.
(let ((sf-1 (make-sndfile "/path/to/sndfile-1.aiff"))))
  (setf (will-be-used sf-1) 11)
  (setf (has-been-used sf-1) 13)
  (print (will-be-used sf-1))
  (print (has-been-used sf-1))
  (reset-usage sf-1)
  (print (will-be-used sf-1))
  (print (has-been-used sf-1)))

=>
11
13
0
0
```

SYNOPSIS:

```
(defmethod reset-usage ((sf sndfile))
```

28.5.603 sndfile/sndfile-ext

[sndfile] [Classes]

NAME:

sndfile-ext

File: sndfile-ext.lsp

```
Class Hierarchy:  named-object -> linked-named-object -> sndfile ->
                  sndfile-ext

Version:          1.0.11

Project:          slippery chicken (algorithmic composition)

Purpose:          Extension of the sndfile class to hold further properties
                  which quantify the character of the sound file, and
                  specifies sound files which can follow the current one.
                  Specifically created to interface with the sndfilenet
                  project in MaxMSP via OSC (see osc.lsp and osc-sc.lsp).

Author:           Michael Edwards: m@michael-edwards.org

Creation date:    16th December 2012, Koh Mak, Thailand

$$ Last modified:  19:18:46 Thu Sep 24 2020 CEST

SVN ID: $Id$
```

28.5.604   sndfile-ext/get-next

[sndfile-ext] [Methods]

DESCRIPTION:

Get the next sound file from the <followers> slot.

ARGUMENTS:

- A sndfile-ext object.

RETURN VALUE:

A sndfile-ext object.

SYNOPSIS:

(defmethod get-next ((sfe sndfile-ext))

28.5.605   sndfile-ext/make-sndfile-ext

[sndfile-ext] [Functions]

DESCRIPTION:

Make a sndfile-ext (extension of sndfile) object which holds the usual
sndfile data as well as a host of others to do with the characteristics of
the sound file. In addition, the followers slot specifies sound files
which can follow the current one. The bitrate, srate, num-frames, and
bytes slots will be filled automatically if the path to an existing sound
file is given.

ARGUMENTS:

See defclass slot descriptions

RETURN VALUE:

A sndfile-ext object or NIL if path is NIL

SYNOPSIS:

```
(defun make-sndfile-ext (path &key id data duration end (start 0.0)
                               (frequency nil) (amplitude 1.0) (cue-num -1)
                               (use t) (pitch -1) (pitch-curve -1)
                               (bandwidth -1)
                               (bandwidth-curve -1) (continuity -1)
                               (continuity-curve -1) (weight -1)
                               (weight-curve -1)
                               (energy -1) (energy-curve -1) (harmonicity -1)
                               (harmonicity-curve -1) (volume -1)
                               (volume-curve -1) (loop-it nil) (bitrate -1)
                               (srate -1) (num-frames -1)
                               (bytes -1) followers group-id)
```

28.5.606   sndfile-ext/max-cue

[sndfile-ext] [Methods]

DESCRIPTION:

Generate the data necessary to preload the sound file in a MaxMSP sflist~
object.

ARGUMENTS:

- The sndfile-ext object.

OPTIONAL ARGUMENTS:

 – Whether to issue an error if the cue-num slot has not been set to a value
   above 1. Default = #'error. Could also be #'warn and NIL.

RETURN VALUE:

 A list of data suitable to be passed via OSC to the sflist~ object.

EXAMPLE:

```
(let* ((sf1 (make-sndfile-ext
              (concatenate 'string
                          cl-user::+slippery-chicken-home-dir+
                          "test-suite/sndfile-1.aiff")
              :cue-num 2 :start 0.3 :end 1.1 :frequency 653)))
  (max-cue sf1))

=>
("preload" 2 "/Users/medward2/lisp/sc/test-suite/sndfile-1.aiff" 300.0 1100.0)
```

SYNOPSIS:

```
(defmethod max-cue ((sfe sndfile-ext) &optional (on-fail #'error))
```

28.5.607   sndfile-ext/max-play

[sndfile-ext] [Methods]

DESCRIPTION:

 Generate the data necessary for MaxMSP to play the sndfile using the
 sflist~ and sfplay~.

 NB fade-dur could be 0 (= no fade)

ARGUMENTS:

 – The sndfile-ext object.
 – The fade (in/out) duration in seconds.
 – The maximum loop duration in seconds.
 – The time to trigger the next file, as a percentage of the current
   sndfile-ext's duration.

RETURN VALUE:

A list of values to be passed via OSC to sndfilenet-aux.maxpat:
cue-number number-of-channels loop speed fade-dururation
fade-out-start-time delay-to-next-snfile-start amplitude

EXAMPLE:

```
(let* ((sf1 (make-sndfile-ext
              (concatenate 'string
                           cl-user::+slippery-chicken-home-dir+
                           "test-suite/sndfile-1.aiff")
              :start 0.3 :end 1.1 :frequency 653)))
  (max-play sf1 20))

=>
(-1 1 0 1.0 0.32000002 0.48)
```

SYNOPSIS:

```
(defmethod max-play ((sfe sndfile-ext) fade-dur max-loop start-next
                     &optional ignore)
```

28.5.608 sndfile-ext/proximity

[sndfile-ext] [Methods]

DESCRIPTION:

In terms of the characteristics expressed in the various class slots,
evaluate the proximity of one sndfile-ext object to another. The closer
they are in character, the closer to 0 the returned value will be. The
order of the two sndfile-ext objects passed to the method is unimportant as
the return value is always >= 0.0. The more slots match, the lower
(closer) the result will be, i.e., slots are only compared if they have a
value >= 0 (they default to -1), so it could be that in one comparison 5/6
slots match exactly, and in another 2/3 match; in both cases the
non-matching slots is off by 1; in that case the first comparison will
return a lower value: 0.067 vs 0.233.

ARGUMENTS:

- first sndile-ext object
- second sndile-ext object

RETURN VALUE:

 A floating point number >= 0.0 where 0.0 indicates an exact match.

EXAMPLE:

```
(let ((sf3 (make-sndfile-ext
            nil :pitch 3 :pitch-curve 4 :bandwidth 10 :energy 2
            :harmonicity-curve 0))
      (sf4 (make-sndfile-ext
            nil :pitch 3 :pitch-curve 4 :bandwidth 10 :energy 2
            :harmonicity-curve 1)))
  ;; harmonicity-curve is slightly different so we get a result > 0
  (print (proximity sf3 sf4))
  (set-characteristic sf4 'harmonicity-curve 0)
  ;; now they're the same so we get 0.0
  (proximity sf3 sf4))

=>
0.12857144
0.0
```

SYNOPSIS:

```
(defmethod proximity ((sfe1 sndfile-ext) (sfe2 sndfile-ext))
```

28.5.609   sndfile-ext/reset

[sndfile-ext] [Methods]

DESCRIPTION:

 Reset the <followers> circular list to the first or any other following
 sound file.

ARGUMENTS:

 A sndfile-ext object.

OPTIONAL ARGUMENTS:

 The position (index) to reset to (will default to 0 i.e. the beginning of
 the list). NB This position may be higher than the number of followers
 attached to any given sndfile-ext object as it will wrap around.

RETURN VALUE:

```
T
```

SYNOPSIS:

```
(defmethod reset ((sfe sndfile-ext) &optional where (warn t))
```

28.5.610 sndfile-ext/set-characteristic

[sndfile-ext] [Methods]

DESCRIPTION:

```
Set the chracteristic of a sndfile-ext object to a given value.  The value
for the slot is first checked to correspond to an accepted range; if not an
error (default) or warning (or nothing) will be issued.
```

ARGUMENTS:

```
- a sndfile-ext object.
- the characteristic, i.e. one of the class slot names, as a symbol.
- the new value, as an accepted integer (see characteristics slot for
  accepted range).
```

OPTIONAL ARGUMENTS:

```
- the function to call if the given value is out of range. Default =
#'error but could also be #'warn, or NIL (if no error message should be
issued).
```

RETURN VALUE:

```
NIL on fail otherwise <value>
```

EXAMPLE:

```
(let ((sf4 (make-sndfile-ext
            nil :pitch 3 :pitch-curve 4 :bandwidth 10 :energy 2
            :harmonicity-curve 1)))
  ;; out of range but no error/warning
  (print (set-characteristic sf4 'harmonicity-curve 15 nil))
  ;; out of range and warn
  (print (set-characteristic sf4 'harmonicity-curve -1 #'warn))
  ;; in range
  (set-characteristic sf4 'harmonicity-curve 0))
```

```
=>
NIL
WARNING:
   sndfile-ext::set-characteristic: No such characteristic: HARMONICITY-CURVE -1
NIL
0
```

SYNOPSIS:

```
(defmethod set-characteristic ((sfe sndfile-ext) characteristic value
                               &optional (on-fail #'error))
```

28.5.611   sndfile/stereo

[sndfile] [Methods]

DESCRIPTION:

 Test whether the CHANNELS slot of a given sndfile object is set to 2.

ARGUMENTS:

 - A sndfile object.

RETURN VALUE:

 Returns T if the CHANNELS slot is set to 2, otherwise returns NIL.

EXAMPLE:

```
;; The method make-sndfile creates a sndfile object with the CHANNELS slot set
;; to NIL. Make a sndfile object, test to see whether the value of the CHANNELS
;; slot is 2; set the CHANNELS slot to 2 and test again.
(let ((sf-1 (make-sndfile "/path/to/sndfile-1.aiff")))
  (print (stereo sf-1))
  (setf (channels sf-1) 2)
  (print (stereo sf-1)))

=>
NIL
T
```

SYNOPSIS:

```
(defmethod stereo ((sf sndfile))
```

28.5.612   linked-named-object/tempo

[linked-named-object] [Classes]

NAME:

 tempo

 File: tempo.lsp

 Class Hierarchy: named-object -> linked-named-object -> tempo

 Version: 1.0.11

 Project: slippery chicken (algorithmic composition)

 Purpose: Implementation of the tempo class which holds very simple
                  tempo information, simply the type of beat and the number
                  of beats per minute etc.

 Author: Michael Edwards: m@michael-edwards.org

 Creation date: March 11th 2001

 $$ Last modified: 17:29:22 Fri Mar 24 2017 GMT

 SVN ID: Id

28.5.613   tempo/make-tempo

[tempo] [Functions]

DESCRIPTION:

 Make a tempo object.

ARGUMENTS:

 - A number indicating beats per minute. If this is >= 10000 we'll treat the
 argument as a usecs slot (number of microseconds quarter note) and calcuate
 the BPM from that. That gives us a maximum BPM of 5999 before we start
 thinking these are usecs (should be fine).

OPTIONAL ARGUMENTS:

 keyword arguments:

- :beat. Sets the "beat" value of the beats per minute; i.e., 'q (or 4) for
  "quarter = xx bpm" etc. Default = 4.
- :id. Sets the ID of the tempo object.
- :description. A text description (string) of the tempo, such as "Allegro
  con brio" etc.

RETURN VALUE:

 A tempo object.

EXAMPLE:

```
;; Default beat is a quarter, thus the following makes a tempo object of
;; quarter=60.
(make-tempo 60)

=>
TEMPO: bpm: 60, beat: 4, beat-value: 4.0, qtr-dur: 1.0
       qtr-bpm: 60.0, usecs: 1000000, description: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: 60

;; Set the beat using the :beat keyword argument. Thus, the following makes a
;; tempo object of dotted-quarter = 96.
(make-tempo 96 :beat 'q.)

;; Add a text description, which is stored in the tempo object's DESCRIPTION
;; slot.
(let ((tt (make-tempo 76 :beat 2 :description "Allegretto")))
  (description tt))

=> "Allegretto"
```

SYNOPSIS:

```
(defun make-tempo (bpm &key (beat 4) id description)
```

28.5.614   tempo/tempo-equal

[tempo] [Methods]

DESCRIPTION:

 Test to determine whether the values of two tempo objects are equal.

ARGUMENTS:

- A first tempo object.
- A second tempo object.

RETURN VALUE:

Returns T if the values of the two tempo objects are equal, otherwise NIL.

EXAMPLE:

```
;; Equal
(let ((tt1 (make-tempo 60))
      (tt2 (make-tempo 60)))
  (tempo-equal tt1 tt2))

=> T

;; Not equal
(let ((tt1 (make-tempo 60))
      (tt2 (make-tempo 96)))
  (tempo-equal tt1 tt2))

=> NIL
```

SYNOPSIS:

```
(defmethod tempo-equal ((t1 tempo) (t2 tempo)))
```

28.5.615   linked-named-object/time-sig

[linked-named-object] [Classes]

NAME:

time-sig

```
File:          time-sig.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist -> time-sig

Version:       1.0.11

Project:       slippery chicken (algorithmic composition)
```

```
 Purpose:            Implementation of a time-sig class that stores
                     information about time signatures, allows comparison of
                     time signatures etc.

 Author:             Michael Edwards: m@michael-edwards.org

 Creation date:      12th February 2001

 $$ Last modified:  16:54:19 Thu Sep 24 2020 CEST

 SVN ID: $Id$
```

28.5.616   time-sig/beat-duration

[time-sig] [Methods]

DESCRIPTION:

 Get the duration in seconds of one beat of the given time-signature at a
 tempo of quarter=60.

ARGUMENTS:

 - A time-sig object.

RETURN VALUE:

 A number.

EXAMPLE:

```
;; Beat duration in seconds for time-signature 2/4 at quarter=60
(let ((ts (make-time-sig '(2 4))))
  (beat-duration ts))

=> 1.0

;; Beat duration in seconds for 6/8 at quarter=60
(let ((ts (make-time-sig '(6 8))))
  (beat-duration ts))

=> 1.5
```

SYNOPSIS:

```
(defmethod beat-duration ((ts time-sig))
```

28.5.617   time-sig/get-beat-as-rhythm

[time-sig] [Methods]

DESCRIPTION:

 Get the beat unit of a given time-sig object and return it as a rhythm.

ARGUMENTS:

 - A time-sig object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to consider the beat of a compound meter to
   be the denominator of the time signature (such as 8 for 6/8) or the beat
   duration derived from the traditionally understood beat of that meter
   (such as Q. for 6/8). NIL = denominator. Default = NIL.

RETURN VALUE:

 A rhythm object.

EXAMPLE:

```
;; Returns a rhythm object
(let ((ts (make-time-sig '(2 4))))
  (get-beat-as-rhythm ts))

=>
RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
        score-rthm: 4.0f0, undotted-value: 4, num-flags: 0, num-dots: 0,
        is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
        is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
        rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
        letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 4, tag: NIL,
data: 4

;; Default for compound meters is to return the denominator of the time
;; signature
(let ((ts (make-time-sig '(6 8))))
  (data (get-beat-as-rhythm ts)))
```

```
=> 8
```

```
;; Setting the optional argument to T returns the compound beat of a compound
;; meter rather than the denominator of the time signature
(let ((ts (make-time-sig '(6 8))))
  (data (get-beat-as-rhythm ts t)))
```

```
=> Q.
```

SYNOPSIS:

```
(defmethod get-beat-as-rhythm ((ts time-sig) &optional handle-compound)
```

28.5.618   time-sig/get-whole-bar-rest

[time-sig] [Methods]

DESCRIPTION:

Create an event object consisting of a rest equal in duration to one full
bar of the given time-sig object.

ARGUMENTS:

 - A time-sig object.

RETURN VALUE:

 Returns an event object.

EXAMPLE:

```
;; Returns an event object
(let ((ts (make-time-sig '(2 4))))
  (get-whole-bar-rest ts))
```

```
=>
EVENT: start-time: NIL, end-time: NIL,
       duration-in-tempo: 0.0,
       compound-duration-in-tempo: 0.0,
       amplitude: 0.7
       bar-num: -1, marks-before: NIL,
       tempo-change: NIL
       instrument-change: NIL
       display-tempo: NIL, start-time-qtrs: -1,
```

```
        midi-time-sig: NIL, midi-program-changes: NIL,
        8va: 0
        pitch-or-chord: NIL
        written-pitch-or-chord: NIL
RHYTHM: value: 2.000, duration: 2.000, rq: 2, is-rest: T,
         score-rthm: 2.0f0, undotted-value: 2, num-flags: 0, num-dots: 0,
         is-tied-to: NIL, is-tied-from: NIL, compound-duration: 2.000,
         is-grace-note: NIL, needs-new-note: NIL, beam: NIL, bracket: NIL,
         rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
         letter-value: 2, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 2, tag: NIL,
data: 2

;; The rhythmic value of the event object returned is equal to the rhythmic
;; duration of a full bar in the given time signature, the PITCH-OR-CHORD slot
;; is set to NIL, and the IS-REST slot is set to T.
(let* ((ts (make-time-sig '(2 4)))
       (tswbr (get-whole-bar-rest ts)))
  (print (value tswbr))
  (print (pitch-or-chord tswbr))
  (print (is-rest tswbr)))

=>
2.0
NIL
T
```

SYNOPSIS:

```
(defmethod get-whole-bar-rest ((ts time-sig))
```

28.5.619   time-sig/is-compound

[time-sig] [Methods]

DESCRIPTION:

Determine whether the value of a given time-sig object is a compound time
signature.

ARGUMENTS:

 - A time-sig object.

RETURN VALUE:

T if the value of the given time-sig object is a compound time signature,
otherwise NIL.

EXAMPLE:

```
;; Testing a time-sig object with a 2/4 time signature returns NIL
(let ((ts (make-time-sig '(2 4))))
  (is-compound ts))

=> NIL

;; Testing a time-sig object with a 6/8 time signature returns T
(let ((ts (make-time-sig '(6 8))))
  (is-compound ts))

=> T
```

SYNOPSIS:

```
(defmethod is-compound ((ts time-sig))
```

28.5.620   time-sig/make-time-sig

[time-sig] [Functions]

DESCRIPTION:

Create a time-sig object. In addition to the numerator and denominator
values, the object also stores other automatically calculated information,
such as whether the signature is simple or compound, the duration of one
bar of the given time signature in seconds, the number of midi-clocks, etc.

ARGUMENTS:

- A two-item list of numbers, the first being the numerator (number of
  beats per measure), the second being the denominator (beat type).

RETURN VALUE:

- A time-sig object.

EXAMPLE:

```
(make-time-sig '(2 4))
```

```
=>
TIME-SIG: num: 2, denom: 4, duration: 2.0, compound: NIL, midi-clocks: 24, num-beats: 2
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0204", tag: NIL,
data: (2 4)
```

SYNOPSIS:

```
(defun make-time-sig (ts)
```

28.5.621   time-sig/make-time-sig-from-duration

[time-sig] [Functions]

DESCRIPTION:

This tries to generate a reasonable time signature from a given duration in
seconds. Will not do anything sophisticated like decide between simple and
compound times.

This will usually result in a numerator >= 2, but tt's not always easy to
decide on the best time-sig, e.g. 2/8 is usually preferable to 1/4, but
1/16 is better than 2/32. Hence there's a call to a routine that
substitutes less-preferable for more-preferable time-sigs.

ARGUMENTS:

- a duration in seconds

OPTIONAL ARGUMENTS:

- the tempo. This will be used to convert the first argument to quarter
  notes

RETURN VALUE:

a time-sig object

EXAMPLE:

```
(make-time-sig-from-duration 2) -> 2/4
(make-time-sig-from-duration 1.75) -> 7/16
```

SYNOPSIS:

```
(defun make-time-sig-from-duration (dur-secs
                                    &optional
                                    (tempo 60.0))
```

28.5.622   time-sig/scale

[time-sig] [Methods]

DESCRIPTION:

 Scale the value of the given time-sig object by a specified factor.

ARGUMENTS:

 - A time-sig object.
 - A number (scaling factor).

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether or not to preserve the meter by maintaining
   the same number of beats as the numerator of the time signature. T =
   preserve the meter. Default = T.

RETURN VALUE:

 A time-sig object.

EXAMPLE:

```
;; Scaling a (2 4) time-sig object by 3 creates a new time-sig object with a
;; value of 6/4
(let ((ts (make-time-sig '(2 4))))
  (scale ts 3))

=>
TIME-SIG: num: 6, denom: 4, duration: 6.0, compound: NIL, midi-clocks: 24, num-beats: 6
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0604", tag: NIL,
data: (6 4)

;; Scaling a (2 4) time-sig object by 2 by default preserves the meter
(let ((ts (make-time-sig '(2 4))))
```

```
  (data (scale ts 2)))
```

```
=> (2 2)
```

```
;; Scaling a (2 4) time-sig object by 2 with the optional argument set to NIL
;; changes the meter and results in a 4/4
(let ((ts (make-time-sig '(2 4))))
  (data (scale ts 2 nil)))
```

```
=> (4 4)
```

```
;; Halving the value of a time-sig object is achieved using a factor of .5
(let ((ts (make-time-sig '(2 4))))
  (data (scale ts .5)))
```

```
=> (2 8)
```

SYNOPSIS:

```
(defmethod scale ((ts time-sig) scaler
                    &optional (preserve-meter t) ignore1 ignore2)
```

28.5.623   time-sig/time-sig-equal

[time-sig] [Methods]

DESCRIPTION:

Determine whether the values of two given time-sig objects are the same. If
they are identical in signature, return T; if they are different signatures
but have the same duration (e.g. 2/4, 4/8, 8/16 etc.) return
TIME-SIG-EQUAL-DURATION; otherwise return NIL.

ARGUMENTS:

- A first time-sig object.
- A second time-sig object.

RETURN VALUE:

Returns T if the time signatures are identical; returns
TIME-SIG-EQUAL-DURATION if they are different signatures with the same
duration; otherwise NIL.

EXAMPLE:

```
;; Two identical signatures return T
(let ((ts1 (make-time-sig '(2 4)))
      (ts2 (make-time-sig '(2 4))))
  (time-sig-equal ts1 ts2))

=> T

;; Two different signatures of the same duration return TIME-SIG-EQUAL-DURATION
(let ((ts1 (make-time-sig '(2 4)))
      (ts2 (make-time-sig '(4 8))))
  (time-sig-equal ts1 ts2))

=> TIME-SIG-EQUAL-DURATION

;; Two completely different signatures return NIL
(let ((ts1 (make-time-sig '(2 4)))
      (ts2 (make-time-sig '(3 4))))
  (time-sig-equal ts1 ts2))

=> NIL
```

SYNOPSIS:

```
(defmethod time-sig-equal ((ts1 time-sig) (ts2 time-sig))
```

29   sc/slippery-chicken

[Classes]

NAME:

```
 slippery-chicken

 File:             slippery-chicken.lsp

 Class Hierarchy:  named-object -> slippery-chicken

 Version:          1.0.11

 Project:          slippery chicken (algorithmic composition)

 Purpose:          Implementation of the slippery-chicken class.

 Author:           Michael Edwards: m@michael-edwards.org
```

```
Creation date:    March 19th 2001

$$ Last modified:  15:35:23 Mon Sep 28 2020 CEST

SVN ID: $Id$
```

29.1   slippery-chicken/auto-set-written

[slippery-chicken] [Methods]

DESCRIPTION:

Automatically set the WRITTEN-PITCH-OR-CHORD slot for all events where the
player plays a transposing instrument (which can change of course).

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :start-bar. NIL or an integer that is the first bar for which the written
  pitch/chord data is to be set. If NIL, start from bar 1. Default = NIL.
- :end-bar. NIL or an integer that is the last bar for which the written
  pitch/chord data is to be set. If NIL, do all bars. Default = NIL.
- :players. NIL, the ID or a list of IDs for the player(s) whose part(s)
  are to be affected. If NIL, all players' parts will be affected.
  Default = NIL.

RETURN VALUE:

 T

EXAMPLE:

```
;;; Create a slippery-chicken object, set all the written-pitch-or-chord ;
;;; slots to NIL and print the results. Apply the method and print the results ;
;;; again to see the difference.           ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
```

```
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                              :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))))))))))
  (next-event mini 'hn nil 1)
  (loop for ne = (next-event mini 'hn)
     while ne
do (setf (written-pitch-or-chord ne) nil))
  (next-event mini 'hn nil 1)
  (print
   (loop for ne = (next-event mini 'hn)
      while ne
collect (written-pitch-or-chord ne)))
  (auto-set-written mini)
  (next-event mini 'hn nil 1)
  (print
   (loop for ne = (next-event mini 'hn)
      while ne
collect (data (written-pitch-or-chord ne)))))

=>
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL)
(C4 D4 E4 FS4 G4 C4 D4 E4 FS4 G4 C4 D4 E4 FS4 G4 C4 D4 E4 FS4 G4 C4 D4 E4
FS4 G4)
```

SYNOPSIS:

```
(defmethod auto-set-written ((sc slippery-chicken) &key start-bar end-bar
                             players)
```

29.2   slippery-chicken/change-bar-line-type

[slippery-chicken] [Methods]

DESCRIPTION:

 Change single to double or repeat bar lines and vice-versa. NB This is a
 score function only, i.e., if you add repeat bar lines these will not (yet)
 be reflected in playback with MIDI or CLM.

ARGUMENTS:

 - the slippery-chicken object
 - the bar number at the end of which you want the bar line to change
 - bar line type: 0 = normal, 1 = double bar, 2 = final double bar, 3 =

    begin repeat, 4 = begin and end repeat, 5 = end repeat. As remembering
    these numbers is a bit arbitrary, you can also just pass symbols, if you
    prefer: 'normal, 'double-bar, 'final-double, 'begin-repeat,
    'begin-end-repeat, 'end-repeat

RETURN VALUE:

 returns the bar-line type

EXAMPLE:

```
(let ((min
       (make-slippery-chicken
        '+minimum+
        :instrument-palette +slippery-chicken-standard-instrument-palette+
        :ensemble '(((fl (flute :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((4 4) - e e e e - - e e e e -)))))
        :rthm-seq-map '((1 ((fl (1)))))))))
;; this piece only has one bar so the bar line will be 2 by default ;
  (print (bar-line-type (get-bar min 1 'fl)))
  (change-bar-line-type min 1 'double-bar)
  (bar-line-type (get-bar min 1 'fl)))
=>
...
2
1
```

SYNOPSIS:

```
(defmethod change-bar-line-type ((sc slippery-chicken) bar-num type)
```

29.3   slippery-chicken/check-beams

[slippery-chicken] [Methods]

DESCRIPTION:

 See the description, with example, for check-beams in rthm-seq-bar.lsp.
 Players can either be a single symbol, a list of symbols, or nil (whereby
 all players will be checked).

SYNOPSIS:

```
(defmethod check-beams ((sc slippery-chicken) &key start-bar end-bar players
                        auto-beam print (on-fail #'warn))
```

29.4 slippery-chicken/check-ties

[slippery-chicken] [Methods]

DESCRIPTION:

 Check that all ties are started and ended properly. If the optional
 argument <same-spellings> is set to T, all tied pitches will be forced to
 have the same enharmonic spellings.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to force all tied pitches to have the same
   enharmonic spellings.
 - a function to call on error, or NIL for no action.

RETURN VALUE:

 T if all tie data is ok, otherwise performs the on-fail function and
 returns NIL.

EXAMPLE:

```
;;; Create a slippery-chicken object, manually create a problem with the ties,
;;; and call check-ties with a #'warn as the on-fail function.
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((cl (b-flat-clarinet :midi-channel 1))))
         :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
         :set-map '((1 (1)))
         :rthm-seq-palette '((1 ((((4 4) { 3 tq tq tq } +q e (s) s)))))
         :rthm-seq-map '((1 ((cl (1)))))))
       (e4 (get-event mini 1 4 'cl)))
  (setf (is-tied-to e4) nil)
  (check-ties mini nil #'warn))

=> WARNING: slippery-chicken::check-ties: bad tie, CL bar 1
```

SYNOPSIS:

```
(defmethod check-ties ((sc slippery-chicken)
                       &optional same-spellings (on-fail #'error))
```

29.5 slippery-chicken/check-time-sigs

[slippery-chicken] [Methods]

DATE:

 28-Jan-2011

DESCRIPTION:

 Check every bar in the given slippery-chicken object to see if all players
 have the same time signature. Drops into the debugger with an error if not.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 T if all players have the same time signature at the same time, otherwise
 drops into the debugger with an error.

EXAMPLE:

```
;; A successful test                       ;
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vn (violin :midi-channel 1))
                      (va (viola :midi-channel 2))
                      (vc (cello :midi-channel 3))))
         :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
         :set-map '((1 (1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) { 3 tq tq tq } +q e (s) s)))))
         :rthm-seq-map '((1 ((vn (1 1 1))
                             (va (1 1 1))
                             (vc (1 1 1)))))))))
  (check-time-sigs mini))

=> T

;; A failing test                    ;
(let* ((mini
        (make-slippery-chicken
         '+mini+
```

```
            :ensemble '(((vn (violin :midi-channel 1))
                         (va (viola :midi-channel 2))
                         (vc (cello :midi-channel 3))))
          :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
          :set-map '((1 (1 1 1)))
          :rthm-seq-palette '((1 ((((4 4) { 3 tq tq tq } +q e (s) s)))))
          :rthm-seq-map '((1 ((vn (1 1 1))
                              (va (1 1 1))
                              (vc (1 1 1)))))))))
  (setf (time-sig (get-bar mini 1 'vn)) '(3 4))
  (check-time-sigs mini))

=>
slippery-chicken::check-time-sigs: time signatures are not the same at bar 1
[Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod check-time-sigs ((sc slippery-chicken))
```

29.6   slippery-chicken/check-tuplets

[slippery-chicken] [Methods]

DESCRIPTION:

Check the qualities of the tuplet brackets in a given slippery-chicken
object to make sure they are all formatted properly (i.e. each starting
tuplet bracket has a closing tuplet bracket etc.). If an error is found,
the method will try to fix it, then re-check, and only issue an error then
if another is found.

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

- The function to use if something is not ok with the tuplets. This
  defaults to #'error, but could also be #'warn for example

RETURN VALUE:

T if all tuplets brackets are ok, otherwise performs the on-fail function
and returns NIL.

EXAMPLE:

```
;;; Create a slippery-chicken object, manually add an error to the tuplet data ;
;;; and call check-tuplets with #'warn as the on-fail function. ;
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((cl (b-flat-clarinet :midi-channel 1))))
         :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
         :set-map '((1 (1)))
         :rthm-seq-palette '((1 ((((4 4) { 3 tq tq tq } +q e (s) s)))))
         :rthm-seq-map '((1 ((cl (1)))))))
       (e1 (get-event mini 1 1 'cl)))
  (setf (bracket e1) nil)
  (check-tuplets mini #'warn))

=> rthm-seq-bar::check-tuplets: Can't close non-existent bracket.
```

SYNOPSIS:

```
(defmethod check-tuplets ((sc slippery-chicken) &optional (on-fail #'error))
```

29.7 slippery-chicken/clm-play

[slippery-chicken] [Methods]

DESCRIPTION:

Using the sound files (samples) defined for the given reference (group ID)
in the sndfile-palette slot of the slippery-chicken object, use CLM to
generate a new sound file using the pitch and timing information of one or
more players' parts from the slippery-chicken object.

NB: The sound file will begin with the first sounding event in the section
    at 0.0 seconds. If there are any leading rests in the player's part,
    these will be omitted in the output file.

By grouping sound files in the sndfile-palette slot, the user can generate
a CLM sound file version of the piece in various 'flavours'; perhaps, for
example, using exclusively source sound files consisting of string samples,
or percussion sounds, or a variety of sounds, as desired. See below for an
example of a sndfile-palette.

By default, as :pitch-synchronous default to NIL (see below) this method
does not attempt to match the pitches of the output sound file to those

generated for the slippery-chicken object, rather, it generates its own
sequence of pitches based on pitches from the current set. Instead of using
the pitches of the specified players' parts--which might produce extreme
sound file transpositions both upwards and downwards--it accesses each note
of the current set (assigned by the set-map to each rthm-seq) from the
bottom up, one voice after another. If do-src is T, transposition will then
be calculated such that the frequency of the sound file (defaults to C4 in
the sndfile object) will be shifted to the pitch of the given pitch of the
set. When the set changes it will play the same respective note number of
that set, etc. This means that, in effect, any given voice will retain the
same pitch for the duration of the rthm-seq (set) but as the sound files
cycle around, and these often have quite different pitch/spectral content,
there's not necessarily a sense of pitch repetition. If you find that this
transposition process still yields some extreme transpositions, the
:note-number keyword can be specified to indicate an index into the current
set of pitches to serve as the lowest voice. If you increase :note-number
the transpositions will get higher. If the number of voices plus this index
exceeds the number of pitches in the set, the method will wrap around to
the lowest pitch of the set. Bear in mind then that when :pitch-synchronous
is NIL, the pitch-seq curves given in the rthm-seq-palette--and therefore
all the carefully chosen pitches in the piece--will be ignored in favour of
representing the transpositions and therefore intervals present in the
set. That might seem like a strange thing to (want to) do, but it's served
me (Michael Edwards) very well on many occasions, so it's still the
default.

If instead of the above method the user would like the pitches in the
resulting sound file to be transposed to match the pitches of the
players' notes, the keyword argument :pitch-synchronous can be set
to T (and do-src should be left set to T as well). This will also work with
chords.

See also make-sfp-from-wavelab-marker-file in sndfile-palette.lsp for
automatically creating a sndfile-palette from markers in a Steinberg
Wavelab marker file.

Event amplitudes are as yet unused by this method.

NB: CLM's nrev instrument must be loaded before calling this method. If
you're using the slippery-chicken app, then this is done for you at
startup.

ARGUMENTS:

 - A slippery chicken object.
 - The ID of the starting section.

- The IDs of the player(s) whose events are to be used to obtain the
  rhythmic structure (and optionally, pitch content) of the resulting sound
  file. This can be a single symbol for an individual player, a list of
  player IDs, or NIL. If NIL, the event from all players' parts will be
  reflected in the output file. Default = NIL.
- The ID of the sound file group in the sndfile-palette slot of the
  slippery-chicken object that contains the source sound files from which
  the new sound file is to be generated.

OPTIONAL ARGUMENTS:

keyword arguments:
- :num-sections. An integer or NIL to indicate how many sections should be
  generated, including the starting section. If NIL, sound file data will
  be generated for all sections of the piece. NB If there are sub-sections
  this will include them in the total, i.e., if section 1 has 3 subsections
  and :num-sections is 2, we'll generate data from the first two
  subsections of section 1, not all subsections of main sections 1 and
  2. Default = NIL.
- :from-sequence. An integer that is the number of the first sequence
  within the specified starting section to be used to generate the output
  file. This argument can only be used when num-sections = 1. Default = 1.
- :num-sequences. NIL or an integer that indicates how many sequences are
  to be generated, including that specified by :from-sequence. If NIL, all
  sequences will be played. This argument can only be used when
  num-sections = 1 and the section has no subsections. Default = NIL.
- :srate. A number that is the sampling rate of the output file
  (independent of the input file). This and the following two arguments
  default to the CLM package globals. See clm.html for more options.
  Default = clm::*clm-srate*.
- :header-type: A CLM package header-type specification to designate the
  output sound file format. For example, clm::mus-riff will produce .wav
  files, clm::mus-aiff will produce .aiff files. The value of this argument
  defaults to the CLM package globals. See clm.html for more
  options. Default = clm::*clm-header-type*.
- :data-format. A CLM package data-format specification to designate the
  output sound file sample data format. For example, clm::mus-float will
  produce a 32-bit little-endian floating-point format; clm::mus-l24int
  will produce little-endian 24-bit integer; mus-bshort will produce 16-bit
  big-endian files, and mus-bfloat will produce 32-bit floating-point
  big-endian files. NB: AIFF and AIFC files are not compatible with little
  endian formats. The value of this argument defaults to the CLM package
  globals. See clm.html for more options.
  Default = clm::*clm-data-format*.
- :sndfile-extension. NIL or a string that will be the extension of the
  output sound file (e.g. ".wav", ".aif"). If NIL, the method will

determine the extension automatically based on the header-type. NB: The
extension does not determine the output sound file format; that is
determined by :header-type. Default = NIL.

- :channels. An integer that is the number of channels in the output sound
  file, limited only by the sound file format specified. Note that both
  stereo and mono sounds from the palette will be randomly panned between
  any two adjacent channels. Default = 2.
- :rev-amt. A number that determines the amount of reverberation for the
  resulting sound file, passed to CLM's nrev.
  NB: 0.1 is a lot. Default = 0.0.
- time-offset. A number that is an offset time in seconds. This produces a
  lead time of a specified number of seconds of silence prior to the sound
  output.
- :play. T or NIL to indicate whether CLM should play the output file
  automatically immediately after it has been written.
  T = play. Default = NIL.
- :inc-start. T or NIL to indicate whether playback of the source sound
  files is to begin at incrementally increasing positions in those files or
  at their respective 0.0 positions every time. If T, the method will
  increment the position in the source sound file from which playback is
  begun such that it reaches the end of the source sound file the last time
  it is 'played'. T = increment start times. Default = NIL.
- :ignore-rests. T or NIL to indicate whether silence should be
  incorporated into the resulting sound file to correspond with rests in
  the player's parts. If T, the sound files will play over the duration of
  rests. However, this is only true on a bar-by-bar basis; i.e., notes at
  the end of one bar will not be continued over into a rest in the next
  bar. This implies that rests at the start of a bar will not be turned
  into sounding notes. T = ignore resets. Default = T.
- :sound-file-palette-ref2. The ID of a sound file group in the given
  slippery-chicken object's sndfile-palette slot. If this reference is
  given, the method will invoke fibonacci-transitions to transition from
  the first specified group of source sound files to this one. If NIL, only
  one group of source sound files will be used. Default = NIL.
- :do-src. T, NIL, a number, or a note-name pitch symbol to indicate whether
  transposition of the source sound files for playback will be calculated
  such that the perceived fundamental frequencies of those sound files are
  shifted to match the pitches of the current set. If do-src is a number
  (frequency in Hertz) or a note-name pitch symbol, the method will use
  only that pitch instead of the sound files' frequencies when transposing
  to the events' pitches. NB Whichever is used, after being converted to a
  sample rate conversion factor, this is always multiplied by the
  src-scaler (see below). T = match sound files' frequencies to set
  pitches. NIL means no transposition will be performed. Default = T.
- :pitch-synchronous: T or NIL to indicate whether the source sound files
  are to be transposed to match the pitches of the events in the given

players' part. This will only be effective if the given source sound file
has a perceptible frequency that has been specified using the sndfile
object's :frequency slot in the sndfile-palette. :do-src must also be T
for this to work. T = match pitches. Default = NIL.

- :reset-snds-each-rs. T or NIL to indicate whether to begin with the first
  source sound file of the specified group at the beginning of each
  rthm-seq. T = begin with the first sound file. Default = T.
- :reset-snds-each-player. T or NIL to indicate whether to begin with the
  first source sound file of the specified group for the beginning of each
  player's part. T = begin with the first sound file. Default = T.
- :play-chance-env. A list of break-point pairs that determines the chance
  that a given event from the source player's part will be reflected in the
  new sound file. It is determined by random selection but uses a fixed
  seed that is re-initialized each time clm-play is called. The following
  default ensures every note will play: '(0 100 100 100).
- :play-chance-env-exp. A number that will be applied as the exponent to
  the play-chance-env's y values to create an exponential interpolation
  between break-point pairs. Default = 0.5.
- :max-start-time. A number that is the last time-point in seconds for
  which events will be processed for the output file. If a maximum start
  time is specified here (in seconds), events after this will be
  skipped. The default value of 99999999 seconds (27778 hours) will result
  in all events being reflected in the sound file.
- :time-scaler. A number that will be the factor by which all start times
  are scaled for the output file (in effect a tempo scaler). If
  :ignore-rests is T, this will also have an indirect effect on
  durations. This argument should not be confused with
  :duration-scaler. Default = 1.0.
- :duration-scaler. A number that is the factor by which the duration of
  all events in the output sound file will be scaled. This does not alter
  start times, and will therefore result in overlapping sounds if greater
  than 1.0. This is not to be confused with :time-scaler. Default = 1.0.
- :normalise. A decimal number that will be the maximum amplitude of the
  resulting output file; i.e., to which the samples will be scaled. Can
  also be NIL, whereupon no normalisation will be performed.
  Default = 0.99
- :amp-env. A list of break-point pairs that will govern the amplitude
  envelope applied to all source-sound files as it is being written to the
  new output file. NB: If the user wants to maintain the original attack of
  the source sound file and is not employing the :inc-start option, this
  should be set to '(0 1 ...). If :inc-start is T, the resulting sound file
  will probably contain clicks from non-zero crossings.
  Default = '(0 0 5 1 60 1 100 0).
- :src-width. An integer that reflects the accuracy of the sample-rate
  conversion. The higher the value, the more accurate the transposition,
  but the slower the processing. Values of 100 might be useful for very low

transpositions. Default = 20.
- :src-scaler: A number that is the factor by which all sample-rate
  conversion values will be scaled (for increasing or decreasing the
  transposition of the overall resulting sound file). Default = 1.0.
- :note-number. A number that is an index, representing the the nth pitch
  of the current set or chord (from the bottom) to be used for the lowest
  player. Default = 0.
- :duration-run-over. T or NIL to indicate whether the method will allow a
  sound file event to extend past the end of specified segment boundaries
  of a sound file in the sndfile-palette. T = allow. Default = NIL.
- :short-file-names. T or NIL to indicate whether abbreviated output file
  names will be automatically created instead of the usually rather long
  names. T = short. Default = NIL.
- :output-name-uniquifier. A user-specified string that will be
  incorporated into the file name, either at the end or the beginning
  depending on whether short-file-names is T or NIL. Default = "".
- :check-overwrite. T or NIL to indicate whether to query the user before
  overwriting existing sound files. T = query. Default = T.
- :print-secs. T or NIL to indicate whether CLM should print the seconds
  computed as it works. T = print. Default = NIL.
- :simulate. T or NIL to indicate whether only the sound file sequencing
  information should be calculated and printed for testing purposes,
  without generating a sound file. T = simulate. Default = NIL.
- :sndfile-palette. NIL or a file name including path and extension that
  contains an external definition of a sndfile-palette. This will replace
  any sndfile-palette defined in the slippery-chicken object. If NIL, the
  one in the slippery-chicken object will be used. Default = NIL.
- :chords. NIL or a list of lists consisting of note-name symbols to be
  used as the pitches for the resulting sound file in place of the pitches
  from the set-map. There must be one chord specified for each sequence. If
  NIL, the pitches from the set-map will be used. Default = NIL.
- :chord-accessor. Sometimes the chord stored in the palette is not a
  simple list of data so we need to access the nth of the chord
  list. Default = NIL.
- :clm-ins. The CLM instrument that should be called to generate sound file
  output. This must accept required and keyword arguments like samp5 (see
  src/samp5.lsp or more simply src/sine.lsp) but can of course choose to
  ignore them. Remember that your CLM instruments will (most probably) be
  in the CLM package so you'll need the clm:: qualifer. Default =
  #'clm::samp5.
- :clm-ins-args. This should be a list of keyword arguments and values to
  pass to each CLM instrument call (e.g. '(:cutoff-freq 2000 :q .6)) or a
  function which takes two arguments (the current event and the event
  number) and returns a list of keyword arguments perhaps based on
  those. Default = NIL.
- :pan-min-max. Each event is panned randomly between any two adjacent

channels in the output sound file. The values used are chosen from the
following list of degrees: (15 25 35 45 55 65 75). However, these can be
scaled to be within new limits by passing a two-element list to
:pan-mix-max. E.g. '(40 50) would limit most values to the middle area of
stereo space. NB if stereo soundfiles are used as input the original
panning will generally be retained. See samp5.lsp for details of how such
files are handled. Default = NIL (i.e. use the list given above).
- :pan-fun. If you want to take charge of selecting pan positions yourself,
  pass a function via this keyword. The function must take one argument: the
  current event, though of course, it can ignore it completely if
  preferred. The function should return a degree value: a number between 0
  and 90. Bear in mind if using samp5 with more than 2 output channels that
  this will still result in the sound moving around multichannel space as
  all you would be setting is the pan value between any two adjacent
  channels. If this isn't your cup of tea, pass a new instrument entirely
  via :clm-ins. Default = NIL.
- :snd-selector. By default the sound files in the given group are cycled
  through, one after the other, returning to the beginning when the end is
  reached. This can be changed by passing a function to :snd-selector. This
  function must take three arguments: the circular-sclist object that
  contains the group's sndfile objects (the sndfiles are in a simple list
  in the data slot); the pitch of the current event (if the event happens
  to be a chord, each pitch of the chord will be handled separately
  i.e. the :snd-selector function will be called for each pitch); and the
  event object. Of course any of these arguments may be ignored by the
  :snd-selector function; indeed it is imagined that the event object will
  be mostly ignored and the pitch object used instead, but it is
  nevertheless passed for completeness. For example, the following
  function, declared on-the-fly using lambda rather than separately with
  defun, makes clm-play act as a traditional sampler would: Assuming our
  sndfile-palette group has a list of sound files whose frequencies really
  correspond to their perceived fundamentals (e.g. see the
  set-frequency-from-filename method, and the kontakt-to-sfp function) we
  choose the nearest sound file in the group to the frequency of the
  current event's pitch:

```
    :snd-selector #'(lambda (sflist pitch event)
                      (declare (ignore event))
                      (get-nearest-by-freq
                       (frequency pitch) (data sflist)))
```

Default = NIL i.e. use the circular selection method.
- :decay-time. Reverb decay time. Default = 3 seconds.
- :snd-transitions. Set the transition from sound-file-ref to sound-file-ref
  over the course of the call to clm-play using a custom envelope instead of
  the default fibonacci-transitions method, e.g. '(0 0 100 1). x-axis is

   arbitrary, y-axis should range from 0 to 1. When reading, y-values will be
   rounded.

RETURN VALUE:

 Total events generated (integer).

EXAMPLE:

```
;;; An example using some of the more frequent arguments ;
(let ((mini
      (make-slippery-chicken
       '+mini+
       :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                    (hn (french-horn :midi-channel 2))
                    (vc (cello :midi-channel 3))))
       :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
       :set-map '((1 (1 1 1 1 1 1 1))
                  (2 (1 1 1 1 1 1 1))
                  (3 (1 1 1 1 1 1 1)))
       :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                                :pitch-seq-palette ((1 (2) 3))))
                           (2 ((((4 4) (q) e (s) s h))
                                :pitch-seq-palette ((1 2 (3)))))
                           (3 ((((4 4) e (s) s h (q)))
                                :pitch-seq-palette ((2 3 3))))
                           (4 ((((4 4) (s) s h (q) e))
                                :pitch-seq-palette ((3 (1) 2)))))
       :rthm-seq-map '((1 ((cl (2 3 2 4 1 3 1))
                           (hn (2 4 1 2 3 1 3))
                           (vc (1 2 2 3 4 1 3))))
                       (2 ((cl (4 2 1 3 3 1 2))
                           (hn (2 1 4 3 2 1 3))
                           (vc (2 3 4 3 1 2 1))))
                       (3 ((cl (3 1 2 4 3 1 2))
                           (hn (3 4 2 1 3 2 1))
                           (vc (3 2 3 1 4 2 1)))))
       :snd-output-dir (get-sc-config 'default-dir)
       :sndfile-palette '(((sndfile-grp-1
                            ((test-sndfile-1.aiff)
                             (test-sndfile-2.aiff)
                             (test-sndfile-3.aiff)))
                           (sndfile-grp-2
                            ((test-sndfile-4.aiff :frequency 834)
                             (test-sndfile-5.aiff)
                             (test-sndfile-6.aiff))))
```

```
                              ("/path/to/sndfiles-dir-1"
                               "/path/to/sndfiles-dir-2")))))
  (clm-play mini 2 '(cl vc) 'sndfile-grp-1
            :num-sections 1
            :srate 48000
            :header-type clm::mus-aiff
            :data-format clm::mus-b24int
            :rev-amt 0.05
            :inc-start t
            :ignore-rests nil
            :sound-file-palette-ref2 'sndfile-grp-2
            :pitch-synchronous t
            :reset-snds-each-rs nil
            :reset-snds-each-player nil))
```

SYNOPSIS:

```
#+clm
(defmethod clm-play ((sc slippery-chicken) section players
                     sound-file-palette-ref
                     &key
                       sound-file-palette-ref2
                       (play-chance-env '(0 100 100 100))
                       (max-start-time 99999999)
                       (play-chance-env-exp 0.5)
                       (time-scaler 1.0)
                       (normalise .99)
                       scaled-by
                       (simulate nil)
                       (from-sequence 1)
                       (num-sequences nil)
                       (num-sections nil)
                       (ignore-rests t)
                       (time-offset 0.0)
                       (chords nil)
                       (chord-accessor nil)
                       (note-number 0)
                       (play nil)
                       (amp-env '(0 0 5 1 60 1 100 0))
                       (inc-start nil)
                       (src-width 20)
                       (src-scaler 1.0)
                       (do-src t)
                       (pitch-synchronous nil)
                       (rev-amt 0.0)
                       (duration-scaler 1.0)
```

```
                        (short-file-names nil)
                        (check-overwrite t)
                        (reset-snds-each-rs t)
                        (reset-snds-each-player t)
                        (duration-run-over nil)
                        (channels 2)
                        (srate clm::*clm-srate*)
                        (header-type clm::*clm-header-type*)
                        (data-format clm::*clm-data-format*)
                        (print-secs nil)
                        (output-name-uniquifier "")
                        (sndfile-extension nil)
                        (sndfile-palette nil)
                        ;; MDE Sat Oct  3 18:45:28 2015 -- for Cameron!
                        pan-fun
                        ;; MDE Thu Oct  1 21:03:59 2015
                        (pan-min-max nil) ; actually '(15 75) by default below
                        ;; MDE Thu Oct  1 19:13:49 2015
                        snd-selector
                        ;; MDE Mon Nov  4 10:10:35 2013 -- the following were
                        ;; added so we could use instruments other than samp5
                        (clm-ins #'clm::samp5)
                        ;; either a list or a function (see above)
                        clm-ins-args
                        ;; DJR Thu 22 Aug 2019 15:08:39 BST
                        ;; clm::with-sound let's us set this, so why doesn't
                        ;; clm-play? Voila!
                        (decay-time 3)
                        ;; DJR Mon 16 Sep 2019 01:26:11 BST
                        ;; We can now set snd-transitions with a custom envelope,
                        ;; e.g. '(0 0 100 1). x-values are arbitrary, y-values
                        ;; are from 0 (for sound-file-palette-ref) and 1 (for
                        ;; sound-file-palette-ref2).
                        snd-transitions)
```

29.8   slippery-chicken/clone

[slippery-chicken] [Methods]

DESCRIPTION:

Copy (clone) the specified instance and all data associated with the
slippery-chicken object.

ARGUMENTS:

- A slippery-chicken object.

RETURN VALUE:

 A slippery-chicken object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :instrument-palette +slippery-chicken-standard-instrument-palette+
        :ensemble '(((fl (flute :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((4 4) - e e e e - - e e e e -)))))
        :rthm-seq-map '((1 ((fl (1)))))))))
  (clone mini))
```

SYNOPSIS:

```
(defmethod clone ((sc slippery-chicken))
```

29.9   slippery-chicken/cmn-display

[slippery-chicken] [Methods]

DESCRIPTION:

 Write the data stored in a given slippery-chicken object to disk as an EPS
 (Encapsulated Postscript) file using CMN.

 Several of the keyword arguments for this method are passed directly to CMN
 and therefore have identical names to CMN functions.

 NB: This might fail if LilyPond files are generated first. If this happens,
     re-evaluate the slippery-chicken object and call cmn-display again.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

```
keyword arguments:
```
- :file. A string that is the directory path with file name and extension
  for the .eps file to be created. Default = "cmn.eps" in the directory
  (get-sc-config 'default-dir) (default "/tmp/")
- :players. NIL or a list of player IDs to indicate whether all players'
  parts should be printed to the score. If NIL, all players' parts will be
  written to the score. If a single symbol or a list of player IDs, only
  the parts of those players will be written to the score. Default = NIL.
- :in-c. T or NIL to indicate whether the output should be printed at
  sounding pitch (in C) or at written pitch. NB: If in C, piccolo and
  double bass maintain their usual octave transpositions.
  T = print at sounding pitch. Default = NIL.
- :respell-notes. T, a list of player IDs paired with a sequence of bar and
  note numbers, or NIL to indicate whether to the cmn-display method should
  call the respell-notes method to the pitches contained in the
  slippery-chicken object according to slippery chicken's enharmonics
  algorithm. If T, the all of the pitches in the object will be considered
  and slippery chicken will convert a number of the pitches to their
  enharmonic equivalents to create more sensible linear pitch progression
  within bars. If a list of player IDs paired with a sequence of bar and
  event numbers is passed, in the form
  '((vln (13 2) (14 3)) (cl (14 3 t))), only the specified pitches are
  changed; e.g., (13 2) = bar 13 note 2 (1-based and counting tied notes
  but not rests). If an additional T is included after the bar number and
  event number (as in the cl example above), only the spelling of the
  written pitch for that event will be changed (the default is to change
  the sounding note spelling only). Chords are not respelled by the default
  algorithm, so if these need to be respelled, this should be indicated by
  sub-grouping the note number portion of the given bar/note pair into a
  2-item sublist, in which the first number is the position of the chord in
  among the attacked notes of that bar and the second number is the
  position of the desired pitch within the chord, counted from the bottom
  up, e.g. (vln (13 (2 1))). If NIL, no changes will be made. Default = T.
- :auto-clefs. T or NIL to indicate whether the cmn-display method should
  call the auto-clefs method, which automatically insert clef changes into
  the parts of those instruments that use more than one clef.
  T = automatically place clef changes. Default = T.
- :start-bar. An integer that indicates the first bar of the object to be
  written to the resulting .eps file. NIL = the first bar. Default = NIL.
- :end-bar. The last bar to be written to the resulting .eps file. NIL =
  the last bar of the slippery-chicken object. Default = NIL.
- :title. T, a string, or NIL to indicate whether to write the title of the
  given slippery-chicken object to the resulting .eps file. If T, the TITLE
  slot of the slippery-chicken object will be used. If a string, the
  specified string will be used instead. If NIL, no title will be included
  in the output. Default = T.

- :size. A number to indicate the overall size of the symbols in the CMN
  output. Default = 15.
- :page-nums. T or NIL to indicate whether page numbers are to be written.
  T = write page numbers. Default = T.
- :empty-staves. T or NIL to indicate whether an empty staff should be
  displayed under each instrument. This can be useful for making editing
  notes by hand. T = print empty staff. Default = NIL.
- :display-sets. T or NIL to indicate whether to print the set of pitches
  used for each rthm-seq on a separate treble-bass grand staff at the
  bottom of each system in the score. T = print. Default = NIL.
- :write-section-info. T or NIL to indicate whether to write the section ID
  into the score. NB: This might not work without first regenerating the
  slippery-chicken object. T = write section IDs. Default = NIL.
- :display-time. T or NIL to indicate whether the elapsed time in
  (mins:secs) should printed above each measure in the resulting score.
  T = print time. Default = NIL.
- :staff-separation. A number that governs the amount of white space to be
  placed between staves, measured in CMN's units. Default = 3.
- :line-separation. A number that governs the amount of white space to be
  placed between systems (i.e. not groups, but a line of music for the
  whole ensemble), measured in CMN's units. Default = 5.
- :group-separation. A number that governs the amount of white space placed
  between groups in a system, measured in CMN's units. Default = 2.
- :system-separation. An indication for how CMN determines the amount of
  white space between systems. If cmn::page-mark, only one system will be
  written per page. Default cmn::line-mark.
- :page-height. A number to indicate the height of the page in centimeters.
  Default = 29.7.
- :page-width. A number to indicate the width of the page in
  centimeters. Default = 21.0.
- :all-output-in-one-file. T or NIL to indicate whether to write a separate
  file for each page of the resulting score. T = write all pages to the
  same multi-page file. Default = T.
- :one-line-per-page. T or NIL to indicate whether to write just one line
  (system) to each page. T = one line per page. Default = NIL.
- :start-bar-numbering. An integer that indicates the number to be given as
  the first bar number in the resulting EPS file. The bars will be numbered
  every five bars starting from this number. NB: The value of this argument
  is passed directly to a CMN function. If a value is given for this
  argument, slippery chicken's own bar-number writing function will be
  disabled. NB: It is recommended that a value not be passed for this
  argument if a value is given for :auto-bar-nums. NIL = bar 1. Default =
  NIL.
- :auto-bar-nums. An integer or NIL to indicate a secondary bar numbering
  interval. This is separate from and in addition to the bar-number written
  in every part every 5 bars. It corresponds to CMN's

automatic-measure-numbers. If set to e.g. 1, a bar number will be printed
for every measure at the top of each system, or if :by-line, a bar number
will be printed at the start of each line. NB: The value of this argument
is passed directly to a CMN function. If a value is given for this
argument, slippery chicken's own bar-number writing function will be
disabled. NB: It is recommended that a value not be passed for this
argument if a value is given for :start-bar-numbering. NIL = no secondary
bar numbering. Default = NIL.

- :rehearsal-letters-all-players. T or NIL to indicate whether rehearsal
  letters should be placed above the staves of all instruments in a score
  (this can be useful when generating parts). If NIL, rehearsal letters are
  only placed above the staves of the instruments at the top of each
  group. T = place rehearsal letters above all instruments. Default = NIL.
- :tempi-all-players. T or NIL to indicate whether to print the tempo above
  all players' parts in the score. T = print above all players' parts.
  Default = NIL.
- :process-event-fun. A user-defined function that takes one argument,
  namely an event object. The specified function will then be called for
  each event in the piece. This could be used, for example, to
  algorithmically add accents, dynamics, or change the colour of notes,
  etc. If NIL, no function will be applied. Default = NIL.
- :automatic-octave-signs. T or NIL to indicate whether ottava signs should
  be inserted automatically when notes would otherwise need many ledger
  lines. T = automatically insert. Default = NIL.
- :multi-bar-rests. T or NIL to indicate whether multiple bars of wrests
  should be consolidated when writing parts. T = consolidate. NIL = write
  each consecutive rest bar separately. Default = NIL.
- :display-marks-in-part. T or NIL to indicate whether to print the marks
  stored in the MARKS-IN-PART slot of each rhythm object in the score. If
  NIL, the indications stored in the MARKS-IN-PART slot are added to parts
  only. T = also print to score. Default = NIL.
- :add-postscript. NIL or postscript code to be added to the .eps file
  after it has been generated. See the add-ps-to-file function for details.
  Default = NIL.
- :auto-open. Whether to open the .EPS file once written. Currently only
  available on OSX with SBCL and CCL. Uses the default app for .EPS
  files, as if opened with 'open' in the terminal. Default = Value of
  (get-sc-config cmn-display-auto-open).

RETURN VALUE:

 Always T.

EXAMPLE:

```
;;; The simplest usage
```

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :title "mini"
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((2 4) (s) (s) e e e))
                                 :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1))))))))
  (cmn-display mini :file "/tmp/mini.eps"))

;;; Used with some of the more frequently implemented keyword arguments
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5)))
                        (2 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5)))
                        (3 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                    (2 (2 2 2 2 2))
                    (3 (3 3 3 3 3)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5))))
                             (2 ((((4 4) q e s s h))
                                  :pitch-seq-palette ((1 2 3 4 5))))
                             (3 ((((4 4) e s s h q))
                                  :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((cl (1 3 2 1 2))
                             (hn (3 1 1 2 2))
                             (vc (1 1 3 2 2))))
                         (2 ((cl (3 1 1 2 2))
                             (hn (1 3 1 2 2))
                             (vc (3 2 2 1 1))))
                         (3 ((cl (1 1 3 2 2))
                             (hn (2 1 1 2 3))
                             (vc (3 1 1 2 2))))))))
  (cmn-display mini
               :file "/tmp/cmn.eps"
               :players '(cl vc)
               :in-c nil
```

```
                        :respell-notes nil
                        :auto-clefs nil
                        :start-bar 8
                        :end-bar 13
                        :title "CMN Fragment"
                        :size 13
                        :page-nums nil
                        :empty-staves t
                        :display-sets t
                        :write-section-info t
                        :display-time t
                        :staff-separation 2
                        :line-separation 3))

=> T
```

SYNOPSIS:

```
#+cmn
(defmethod cmn-display ((sc slippery-chicken)
                        &key
                        (respell-notes t)
                        (start-bar nil)
                        (start-bar-numbering nil)
                        (end-bar nil)
                        ;; MDE Fri Apr  6 13:27:08 2012
                        (title t)
                        (file (format nil "~a~a.eps"
                                      (get-sc-config 'default-dir)
                                      (filename-from-title (title sc))))
                        (all-output-in-one-file t)
                        (one-line-per-page nil)
                        (staff-separation 3)
                        (line-separation 5)
                        (empty-staves nil)
                        (write-section-info nil)
                        (group-separation 2)
                        (system-separation cmn::line-mark)
                        (process-event-fun nil)
                        (display-sets nil)
                        (rehearsal-letters-all-players nil)
                        (display-marks-in-part nil)
                        (tempi-all-players nil)
                        (players nil)
                        (page-height 29.7)
                        (page-width 21.0)
```

```
                         (size 15)
                         (auto-bar-nums nil)
                         (page-nums t)
                         (in-c nil)
                         (auto-clefs t)
                         (multi-bar-rests nil)
                         (automatic-octave-signs nil)
                         (display-time nil)
                         (auto-open (get-sc-config 'cmn-display-auto-open))
                         (add-postscript nil))
```

29.10   slippery-chicken/copy-bars

[slippery-chicken] [Methods]

DESCRIPTION:

Copy the rhythmic contents (rthm-seq-bar objects) from the specified bars
of one specified player's part to another. NB No check is performed to
ensure that the copied notes are within the new instrument's range.

ARGUMENTS:

 - A slippery-chicken object.
 - A 1-based integer or assoc-list reference (section seq-num bar-num) that
   is the number of the first bar in the source player's part whose rhythmic
   contents are to be copied.
 - A 1-based integer or assoc-list reference (section seq-num bar-num) that
   is the number of the first bar in the target player's part to which the
   rhythmic contents are to be copied.
 - The ID of the source player's part.
 - The ID of the target player's part.
 - NIL or an integer that is the number of bars to copy, including the
   start-bar. When NIL, all bars in the piece starting from <to-start-bar>
   will be copied.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether to print feedback to the listener about the
   copying process. T = print. Default = NIL.

RETURN VALUE:

 T

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((hn (french-horn :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5))))
                            (2 ((((4 4) h h))
                                :pitch-seq-palette ((1 2)))))
        :rthm-seq-map '((1 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((hn (2 2 2 2 2))
                            (vc (2 2 2 2 2))))
                        (3 ((hn (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (copy-bars mini 7 2 'vc 'hn 2 t))
```

=> T

SYNOPSIS:

```
(defmethod copy-bars ((sc slippery-chicken) from-start-bar to-start-bar
                      from-player to-player num-bars
                      &optional (print-bar-nums nil))
```

29.11   slippery-chicken/count-microtones

[slippery-chicken] [Methods]

DATE:

 January 30th 2017

DESCRIPTION:

 Count the number of microtonal pitches in the slippery-chicken object.

ARGUMENTS:

- the slippery-chicken object

OPTIONAL ARGUMENTS:

keyword arguments:
- :players. the players for whom the count should be executed; either a
  single player symbol or a list of symbols
- :start-bar. The bar number in which the counting should begin. Default =
  1
- :end-bar. The bar number in which counting should end. Default = NIL
  which means count through the final bar.

RETURN VALUE:

Four values: the number of microtones, the number of non-microtones, the
total number of pitches, the percentage of microtonal pitches.

Note that the total number of pitches will almost certainly be different
from the number of events as some events will be rests, others single pitch
notes, yet others chords.

EXAMPLE:

```
(count-microtones +jitterbug+)
1420
2312
3732
38.049305
```

SYNOPSIS:

```
(defmethod count-microtones ((sc slippery-chicken) &key players (start-bar 1)
                                                 end-bar)
```

29.12   slippery-chicken/count-notes

[slippery-chicken] [Methods]

DESCRIPTION:

Returns the number of notes between start-bar and end-bar (both
inclusive).

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the first bar in which notes will be counted. If NIL,
  then 1
- An integer that is the last bar in which notes will be counted. If NIL,
  then the last bar of the piece.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to count just the number of attacked notes
  (not including ties) or the number of note events (including ties).
  T = just attacked notes. Default = NIL.
  NB: A chord counts as one note only.
- NIL or a list of one or more IDs of the players whose notes should be
  counted. This can be a single symbol or a list of players. If NIL, the
  notes in all players' parts will be counted. Default = NIL.

RETURN VALUE:

 An integer that is the number of notes.

EXAMPLE:

```
;;; Using defaults
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s)
                                 (q (e) s +s h)
                                 ((e) s (s) (q) h))
                                :pitch-seq-palette ((1 2 3 4 5 1 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (count-notes mini 2 11))

=> 62
```

```
;;; Counting all notes just for player 'vc ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s)
                                 (q (e) s +s h)
                                 ((e) s (s) (q) h))
                                :pitch-seq-palette ((1 2 3 4 5 1 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (count-notes mini 2 11 nil 'vc))

=> 31

;;; Counting just the attacked notes for player 'vc ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s)
                                 (q (e) s +s h)
                                 ((e) s (s) (q) h))
                                :pitch-seq-palette ((1 2 3 4 5 1 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (count-notes mini 2 11 t 'vc))
```

```
=> 27
```

SYNOPSIS:

```
(defmethod count-notes ((sc slippery-chicken) start-bar end-bar
                        &optional just-attacks players)
```

29.13   slippery-chicken/empty-bars?

[slippery-chicken] [Methods]

DESCRIPTION:

 Test whether players have rest/empty bars within a given range.

ARGUMENTS:

 - the slippery-chicken object
 - the start bar (integer, inclusive)
 - the end bar (integer, inclusive)

OPTIONAL ARGUMENTS:

 - the players we want to test. Either a list of player IDs, a single ID, or
   nil if all players should be tested. Default = NIL = all players.
 - T or NIL to indicate whether a bar of only rests (instead of a full bar
   rest) should be considered empty. Default = NIL.

RETURN VALUE:

 T if all players have rest bars within the given range, NIL if no.

SYNOPSIS:

```
(defmethod empty-bars? ((sc slippery-chicken) start-bar end-bar
                        &optional players all-rests)
```

29.14   slippery-chicken/find-boundaries

[slippery-chicken] [Methods]

DESCRIPTION:

This methods tries to find structural boundaries (i.e section starts) in a
complete piece. It does so by looking purely at event densities across bars
and groups of bars; when these change sharply, a section change is noted.

ARGUMENTS:

- the slippery-chicken object to analyse

OPTIONAL ARGUMENTS:

- when looking at groups of bars, how many bars at a time? Default = 3
- when looking at the sharpness of the envelope, what percentage change in
  number of events per bar (based in the min/max in the overall envelope)
  is considered a steep enough change. Default = 50 (%)

RETURN VALUE:

A list of bar numbers

SYNOPSIS:

```
(defmethod find-boundaries ((sc slippery-chicken) &optional
                            (sum-bars 3) (jump-threshold 50))
```

29.15 slippery-chicken/find-end-tie

[slippery-chicken] [Methods]

DATE:

November 8th 2018, Heidhausen

DESCRIPTION:

Find the event which is the last in a group of tied notes.

ARGUMENTS:

- the slippery-chicken object
- the event object at the start or in the middle of a the group of tied
  notes

OPTIONAL ARGUMENTS:

- the function, generally some kind of error, to call if the given event is
  not tied from. Default = #'error.

RETURN VALUE:

the last event object which is tied to or NIL if the given event is not tied
from.

SYNOPSIS:

```
(defmethod find-end-tie ((sc slippery-chicken) (e event)
                         &optional (on-fail #'error))
```

29.16   slippery-chicken/find-note

[slippery-chicken] [Methods]

DATE:

 09-Apr-2011

DESCRIPTION:

 Print to the Listener the numbers of all bars in a specified player's part
 of a given slippery-chicken object in which the specified pitch is
 found. Note that pitch symbols rather than objects are searched for so
 that small microtonal variations are be ignored.

ARGUMENTS:

 - A slippery-chicken object.
 - A player ID.
 - A note-name pitch symbol or a list of note-name pitch symbols for the
   pitch to be sought. If a list, this will be handled as a chord.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :written. T or NIL to indicate whether to look for the specified pitch as
   as a written note only. T = as written only. Default = NIL.
 - :start-bar. An integer that is the first bar in which to search for the
   given pitch. This number is inclusive. Default = 1.
 - :end-bar. An integer that is the last bar in which to search for the
   given pitch. This number is inclusive. Default = number of bars in the
   given slippery-chicken object.

RETURN VALUE:

 Returns a list of all corresponding event objects found. Prints the bar
 numbers of the results directly to the Lisp listener.

EXAMPLE:

```
;;; Prints the bar number for all occurrences in the entire piece by default ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s)
                                 (q (e) s +s h)
                                 ((e) s (s) (q) h))
                                :pitch-seq-palette ((1 2 3 4 5 1 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))))))
  (find-note mini 'vc 'f4))

=>
bar 1
bar 3
bar 4
bar 6
bar 7
bar 9
bar 10
bar 12
bar 13
bar 15
bar 16
bar 18
bar 19
bar 21
bar 22
bar 24
```

```
bar 25
bar 27
bar 28
bar 30
bar 31
bar 33
bar 34
bar 36
bar 37
bar 39
bar 40
bar 42
bar 43
bar 45


;;; Examples of use specifying the optional arguments ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s)
                                 (q (e) s +s h)
                                 ((e) s (s) (q) h))
                                :pitch-seq-palette ((1 2 3 4 5 1 3 2)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))) 
  (find-note mini 'cl 'f3)
  (find-note mini 'cl 'f3 :written t)
  (find-note mini 'vc 'f4 :start-bar 3 :end-bar 17))
```

SYNOPSIS:

```
(defmethod find-note ((sc slippery-chicken) player note &key (written nil)
                      start-bar end-bar)
```

29.17   slippery-chicken/find-rehearsal-letters

[slippery-chicken] [Methods]

DESCRIPTION:

Return in list form the numbers of bars in the given slippery-chicken
object that have rehearsal letters.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 A list of numbers.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :rehearsal-letters '(2 5 7)
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) (s) (s) e e e))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1 1 1))))))))
  (find-rehearsal-letters mini))

=> (2 5 7)
```

SYNOPSIS:

```
(defmethod find-rehearsal-letters ((sc slippery-chicken))
```

29.18   slippery-chicken/get-all-events

[slippery-chicken] [Methods]

DATE:

 August 22nd 2013 (Edinburgh)

DESCRIPTION:

 Return a flat list containing all the events of a slippery-chicken object
 fo the given player.

ARGUMENTS:

 - The slippery-chicken object
 - The player (symbol)

RETURN VALUE:

 A list of event objects

SYNOPSIS:

```
(defmethod get-all-events ((sc slippery-chicken) player)
```

29.19   slippery-chicken/get-all-section-refs

[slippery-chicken] [Methods]

DESCRIPTION:

 Return all section IDs as a list of lists. Subsection IDs will be contained
 in the same sublists as their enclosing sections.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 A list of lists.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax (alto-sax :midi-channel 1))))
        :set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
        :set-map '((1 (1 1 1))
                   (2 (1 1 1))
```

```
                        (3 ((a (1 1 1))
                            (b ((x (1 1 1))
                                (y (1 1 1)))))))
                        (4 ((a (1 1 1))
                            (b (1 1 1))
                            (c (1 1 1 1))))
                        (5 (1 1 1))
                        (6 (1 1 1))
                        (7 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                               :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1))))
                        (2 ((sax (1 1 1))))
                        (3 ((a ((sax (1 1 1))))
                            (b ((x ((sax (1 1 1))))
                                (y ((sax (1 1 1))))))))
                        (4 ((a ((sax (1 1 1))))
                            (b ((sax (1 1 1))))
                            (c ((sax (1 1 1 1))))))
                        (5 ((sax (1 1 1))))
                        (6 ((sax (1 1 1))))
                        (7 ((sax (1 1 1)))))))
  (get-all-section-refs mini))

=> ((1) (2) (3 A) (3 B X) (3 B Y) (4 A) (4 B) (4 C) (5) (6) (7))
```

SYNOPSIS:

```
(defmethod get-all-section-refs ((sc slippery-chicken))
```

29.20 slippery-chicken/get-bar

[slippery-chicken] [Methods]

DESCRIPTION:

Get the rthm-seq-bar object located at a specified bar number within a
given player's part.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar within the overall piece for
  which the rthm-seq-bar object is sought.
- The ID of the player from whose part the rthm-seq-bar object is

sought. If this is passed as NIL, the method will return the rthm-seq-bar
objects for all players in the ensemble at the specified bar number.
NB: Although listed as an optional argument, the player ID is actually
required. It is listed as optional due to method inheritance.

OPTIONAL ARGUMENTS:

 - (see the comment on the <player> argument above.

RETURN VALUE:

 A rthm-seq-bar object (or objects).

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s)
                                 (q e s s h)
                                 (e s s q h))
                                :pitch-seq-palette ((1 2 3 4 5
                                                     1 3 2 4 5
                                                     3 5 2 4 1)))))
        :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (2 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1))))
                        (3 ((cl (1 1 1 1 1))
                            (vc (1 1 1 1 1)))))))))
  (get-bar mini 17 'cl))

=>
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: NIL, bar-num: 17,
old-bar-nums: NIL, write-bar-num: NIL, start-time: 64.000,
start-time-qtrs: 64.0, is-rest-bar: NIL, multi-bar-rest: NIL,
show-rest: T, notes-needed: 5,
tuplets: NIL, nudge-factor: 0.35, beams: NIL,
current-time-sig: 2, write-time-sig: NIL, num-rests: 0,
```

```
num-rhythms: 5, num-score-notes: 5, parent-start-end: NIL,
missing-duration: NIL, bar-line-type: 0,
player-section-ref: (2 CL), nth-seq: 0, nth-bar: 1,
rehearsal-letter: NIL, all-time-sigs: (too long to print)
sounding-duration: 4.000,
rhythms: (
[...]
```

SYNOPSIS:

```
(defmethod get-bar ((sc slippery-chicken) bar-num &optional player)
```

29.21   slippery-chicken/get-bar-from-ref

[slippery-chicken] [Methods]

DESCRIPTION:

Return a rthm-seq-bar object from the piece by specifying its section,
sequence number, bar number, and the player. Sequenz-num and bar-num are
1-based.

ARGUMENTS:

- A slippery-chicken object.
- A section ID (number or list).
- A player ID.
- An integer that is the number of the sequence in the section from which
  the bar is to be returned (1-based).
- An integer that is the number of the bar within the given sequence
  (1-based).

RETURN VALUE:

A rthm-seq-bar object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1))
```

```
                         (2 (1 1 1 1 1))
                         (3 (1 1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) h q e s s)
                                   (q e s s h)
                                   (e s s q h))
                                  :pitch-seq-palette ((1 2 3 4 5
                                                       1 3 2 4 5
                                                       3 5 2 4 1)))))
         :rthm-seq-map '((1 ((cl (1 1 1 1 1))
                             (vc (1 1 1 1 1))))
                         (2 ((cl (1 1 1 1 1))
                             (vc (1 1 1 1 1))))
                         (3 ((cl (1 1 1 1 1))
                             (vc (1 1 1 1 1)))))))))
  (get-bar-from-ref mini 2 'vc 3 2))
=>
RTHM-SEQ-BAR: time-sig: 2 (4 4), time-sig-given: NIL, bar-num: 23,
old-bar-nums: NIL, write-bar-num: NIL, start-time: 88.000,
start-time-qtrs: 88.0, is-rest-bar: NIL, multi-bar-rest: NIL,
show-rest: T, notes-needed: 5,
tuplets: NIL, nudge-factor: 0.35, beams: NIL,
current-time-sig: 2, write-time-sig: NIL, num-rests: 0,
num-rhythms: 5, num-score-notes: 5, parent-start-end: NIL,
missing-duration: NIL, bar-line-type: 0,
player-section-ref: (2 VC), nth-seq: 2, nth-bar: 1,
rehearsal-letter: NIL, all-time-sigs: (too long to print)
sounding-duration: 4.000,
rhythms: (
[...]
```

SYNOPSIS:

```
(defmethod get-bar-from-ref ((sc slippery-chicken) section player
                             sequenz-num bar-num)
```

29.22   slippery-chicken/get-bar-num-from-ref

[slippery-chicken] [Methods]

DESCRIPTION:

 Get the bar number of a given rthm-seq-bar object by specifying the
 section, sequenz, and number of the bar within that sequenz.

ARGUMENTS:

  - A slippery-chicken object.
  - The ID of the section in which the given rthm-seq-bar object is located.
  - An integer that is the number of the sequence within that section in
    which the rthm-seq-bar object is located.
  - The number of the bar within the given rthm-seq-bar object for which the
    overall bar number (within the entire piece) is sought.

RETURN VALUE:

 An integer.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((cl (b-flat-clarinet :midi-channel 1))
(vc (cello :midi-channel 2))))
:set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
:set-map '((1 (1 1 1 1 1))
(2 (1 1 1 1 1))
(3 (1 1 1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s)
(q e s s h)
(e s s q h))
:pitch-seq-palette ((1 2 3 4 5
1 3 2 4 5
3 5 2 4 1)))))
:rthm-seq-map '((1 ((cl (1 1 1 1 1))
(vc (1 1 1 1 1))))
(2 ((cl (1 1 1 1 1))
(vc (1 1 1 1 1))))
(3 ((cl (1 1 1 1 1))
(vc (1 1 1 1 1)))))))))
(get-bar-num-from-ref mini 2 4 3))

=> 27
```

SYNOPSIS:

```
(defmethod get-bar-num-from-ref ((sc slippery-chicken) section
                                 sequenz-num bar-num)
```

29.23   slippery-chicken/get-clef

[slippery-chicken] [Methods]

DATE:

 11-Apr-2011

DESCRIPTION:

 Get the clef symbol attached to a specified event.

 NB: The very first clef symbol in the very first measure of a given
     player's part is determined by the corresponding instrument object and
     attached to differently; as such, it cannot be retrieved using this
     method.
 NB: All clef symbols after the starting clef are added using the auto-clefs
     method, either directly or by default in the cmn-display or
     write-lp-data-for-all methods.

ARGUMENTS:

 - A slippery-chicken object.
 - (NB: The optional arguments are actually required.)

OPTIONAL ARGUMENTS:

 NB: The optional arguments are actually required.
 - An integer that is the number of the bar from which to return the clef
   symbol.
 - An integer that is the number of the event object within that bar from
   which to retrieve the clef symbol.
 - The ID of the player from whose part the clef symbol is to be returned.

RETURN VALUE:

 A clef symbol.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((vc (cello :midi-channel 1))))
:tempo-map '((1 (q 96)))
:set-palette '((1 ((g2 f4 e5))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((5 4) e e e e e e e e e e))
:pitch-seq-palette ((1 1 2 2 2 2 3 3 3 1)))))
```

```
:rthm-seq-map '((1 ((vc (1 1 1))))))))))
(auto-clefs mini)
(get-clef mini 1 3 'vc))

=> TENOR
```

SYNOPSIS:

```
(defmethod get-clef ((sc slippery-chicken) &optional bar-num event-num player)
```

29.24 slippery-chicken/get-current-instrument-for-player

[slippery-chicken] [Methods]

DESCRIPTION:

Get the currently active instrument for a given player in a specified
sequence of a slippery-chicken object, as defined in the
instrument-change-map.

ARGUMENTS:

- The ID of the section from which to retrieve the current instrument for
  the specified player. This can also be a reference, e.g. in the form
  '(2 1).
- The ID of the player for whom the current instrument is sought.
- The number of the sequence from which to retrieve the current
  instrument. This is a 1-based number. A slippery-chicken object.

RETURN VALUE:

An instrument object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                     (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1)))
```

```
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                  :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))))))))
  (get-current-instrument-for-player 2 'sax 3 mini))

=>
INSTRUMENT: lowest-written: BF3, highest-written: FS6
lowest-sounding: CS3, highest-sounding: A5
starting-clef: TREBLE, clefs: (TREBLE), clefs-in-c: (TREBLE)
prefers-notes: NIL, midi-program: 66
transposition: EF, transposition-semitones: -9
score-write-in-c: NIL, score-write-bar-line: NIL
chords: NIL, chord-function: NIL,
total-bars: 5 total-notes: 25, total-duration: 20.000
total-degrees: 2920, microtones: T
missing-notes: (BQF3 BQF4), subset-id: NIL
staff-name: alto saxophone, staff-short-name: alt sax,

largest-fast-leap: 999, tessitura: BQF3
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: ALTO-SAX, tag: NIL,
data: NIL
```

SYNOPSIS:

```
(defmethod get-current-instrument-for-player (section player sequence
                                              (sc slippery-chicken))
```

29.25   slippery-chicken/get-event

[slippery-chicken] [Methods]

DESCRIPTION:

 Retrieve a specified event object from a slippery-chicken object, giving
 bar number, event number, and player.

 NB: This counts returns event objects, regardless of whether they are notes
     or rests.

ARGUMENTS:

 - A slippery-chicken object.

- An integer that is the number of the bar from which the event object is
  to be returned.
- An integer that is the number of the event object to be returned from
  that bar. This number is 1-based and counts all events, including notes,
  rests, and tied notes.
- A symbol name for the player.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether an error should be signalled if the event
  doesn't exist.

RETURN VALUE:

 An event object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))
                     (vc (cello :midi-channel 2))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) (e) e+e. 32 (32)))
                                :pitch-seq-palette (((1) 2)))))
        :rthm-seq-map '((1 ((vn (1 1 1))
                            (vc (1 1 1)))))))))
  (get-event mini 2 4 'vn))
```

```
=>
EVENT: start-time: 3.750, end-time: 3.875,
duration-in-tempo: 0.125,
compound-duration-in-tempo: 0.125,
amplitude: 0.700
bar-num: 2, marks-before: NIL,
tempo-change: NIL
instrument-change: NIL
display-tempo: NIL, start-time-qtrs: 3.750,
midi-time-sig: NIL, midi-program-changes: NIL,
8va: 0
pitch-or-chord:
PITCH: frequency: 293.665, midi-note: 62, midi-channel: 1
```

```
pitch-bend: 0.0
degree: 124, data-consistent: T, white-note: D4
nearest-chromatic: D4
src: 1.122462, src-ref-pitch: C4, score-note: D4
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: NIL, natural: T,
octave: 4, c5ths: 0, no-8ve: D, no-8ve-no-acc: D
show-accidental: T, white-degree: 29,
accidental: N,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: D4, tag: NIL,
data: D4
**************

written-pitch-or-chord: NIL
RHYTHM: value: 32.000, duration: 0.125, rq: 1/8, is-rest: NIL,
score-rthm: 32.0, undotted-value: 32, num-flags: 3, num-dots: 0,
is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.125,
is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
letter-value: 32, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: 32, tag: NIL,
data: 32
```

SYNOPSIS:

```
(defmethod get-event ((sc slippery-chicken) bar-num event-num player
                      &optional (error t))
```

29.26   slippery-chicken/get-events-from-to

[slippery-chicken] [Methods]

DATE:

 22-Jul-2011 (Pula)

DESCRIPTION:

 Return a list of event objects for a given player, specifying the region by
 bar and event number.

ARGUMENTS:

- A slippery-chicken object.
- A player ID.
- An integer (1-based) that is the first bar from which to return events.
- An integer (1-based) that is the first event object in the start-bar to
  return.
- An integer (1-based) that is the last bar from which to return events.

OPTIONAL ARGUMENTS:

- An integer (1-based) that is the last event within the end-bar to
  return.

RETURN VALUE:

 A flat list of event objects.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                     (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1)))))))))
  (get-events-from-to mini 'sax 3 2 5 3))

=>
(
EVENT: start-time: 10.000, end-time: 11.000,
duration-in-tempo: 1.000,
compound-duration-in-tempo: 1.000,
amplitude: 0.700
bar-num: 3, marks-before: NIL,
tempo-change: NIL
instrument-change: NIL
```

```
display-tempo: NIL, start-time-qtrs: 10.000,
midi-time-sig: NIL, midi-program-changes: NIL,
8va: 0
pitch-or-chord:
PITCH: frequency: 164.814, midi-note: 52, midi-channel: 1
pitch-bend: 0.0
degree: 104, data-consistent: T, white-note: E3
nearest-chromatic: E3
src: 0.62996054, src-ref-pitch: C4, score-note: E3
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: NIL, flat: NIL, natural: T,
octave: 3, c5ths: 0, no-8ve: E, no-8ve-no-acc: E
show-accidental: T, white-degree: 23,
accidental: N,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E3, tag: NIL,
data: E3
**************

written-pitch-or-chord:
PITCH: frequency: 369.994, midi-note: 66, midi-channel: 1
pitch-bend: 0.0
degree: 132, data-consistent: T, white-note: F4
nearest-chromatic: FS4
src: 1.4142135, src-ref-pitch: C4, score-note: FS4
qtr-sharp: NIL, qtr-flat: NIL, qtr-tone: NIL,
micro-tone: NIL,
sharp: T, flat: NIL, natural: NIL,
octave: 4, c5ths: 1, no-8ve: FS, no-8ve-no-acc: F
show-accidental: T, white-degree: 31,
accidental: S,
accidental-in-parentheses: NIL, marks: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: FS4, tag: NIL,
data: FS4
**************

RHYTHM: value: 4.000, duration: 1.000, rq: 1, is-rest: NIL,
score-rthm: 4.0, undotted-value: 4, num-flags: 0, num-dots: 0,
is-tied-to: NIL, is-tied-from: NIL, compound-duration: 1.000,
is-grace-note: NIL, needs-new-note: T, beam: NIL, bracket: NIL,
rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
letter-value: 4, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
```

```
NAMED-OBJECT: id: Q, tag: NIL,
data: Q
**************
```

```
EVENT: start-time: 11.000, end-time: 11.500,
[...]
```

SYNOPSIS:

```
(defmethod get-events-from-to ((sc slippery-chicken) player start-bar
                                start-event end-bar &optional end-event)
```

29.27   slippery-chicken/get-events-sorted-by-time

[slippery-chicken] [Methods]

DESCRIPTION:

 Get all the events of all players between the given bars and then order
 them by ascending time.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :start-bar. An integer that is the first bar in which the function is to
   be applied to event objects. Default = 1.
 - :end-bar. NIL or an integer that is the last bar for which the event
   objects should be returned. If NIL, the last bar of the
   slippery-chicken object is used. Default = NIL.

RETURN VALUE:

 A list of event objects.

SYNOPSIS:

```
(defmethod get-events-sorted-by-time ((sc slippery-chicken)
                                       &key (start-bar 1) end-bar)
```

29.28 slippery-chicken/get-instrument-for-player-at-bar

[slippery-chicken] [Methods]

DATE:

 09-Feb-2011

DESCRIPTION:

 Get the current instrument for a specified player at a specified bar number
 in a slippery-chicken object, as defined in the instrument-change-map.

ARGUMENTS:

 - The ID of a player in the slippery-chicken object.
 - An integer that is the number of the bar from which to get the current
   instrument.
 - A slippery-chicken object.

RETURN VALUE:

 An instrument object.

EXAMPLE:

```
(let* ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                      (db (double-bass :midi-channel 2))))
         :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                  (2 ((sax ((2 alto-sax) (5 tenor-sax))))))
         :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
         :set-map '((1 (1 1 1 1 1))
                    (2 (1 1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5)))))
         :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                             (db (1 1 1 1 1))))
                         (2 ((sax (1 1 1 1 1))
                             (db (1 1 1 1 1)))))))))
  (get-instrument-for-player-at-bar 'sax 3 mini))

=>
```

```
INSTRUMENT: lowest-written: BF3, highest-written: FS6
lowest-sounding: AF2, highest-sounding: E5
starting-clef: TREBLE, clefs: (TREBLE), clefs-in-c: (BASS TREBLE)
prefers-notes: NIL, midi-program: 67
transposition: BF, transposition-semitones: -14
score-write-in-c: NIL, score-write-bar-line: NIL
chords: NIL, chord-function: NIL,
total-bars: 5 total-notes: 25, total-duration: 20.000
total-degrees: 2710, microtones: T
missing-notes: (FQS3 FQS4), subset-id: NIL
staff-name: tenor sax, staff-short-name: ten sax,

largest-fast-leap: 999, tessitura: FS3
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: TENOR-SAX, tag: NIL,
data: NIL
```

SYNOPSIS:

```
(defmethod get-instrument-for-player-at-bar (player bar (sc slippery-chicken))
```

29.29   slippery-chicken/get-last-bar

[slippery-chicken] [Methods]

DATE:

 September 28th 2020, Heidhausen

DESCRIPTION:

 Get the last bars for each of the players

ARGUMENTS:

 - the slippery-chicken object

RETURN VALUE:

 a list of rthm-seq-bar objects

SYNOPSIS:

```
(defmethod get-last-bar ((sc slippery-chicken))
```

29.30   slippery-chicken/get-last-event

[slippery-chicken] [Methods]

DATE:

 September 28th 2020, Heidhausen

DESCRIPTION:

 Get the last events (of the last bars) for each of the players

ARGUMENTS:

 - the slippery-chicken object

RETURN VALUE:

 a list of event objects

SYNOPSIS:

(defmethod get-last-event ((sc slippery-chicken))

29.31   slippery-chicken/get-nearest-event

[slippery-chicken] [Methods]

DESCRIPTION:

 Find the nearest event by time to <reference-event> in the <search-player>.

ARGUMENTS:

 - the slippery-chicken object
 - the reference event object that we want to find the nearest event to in
   another player
 - the player we want to find the nearest event in (symbol).

OPTIONAL ARGUMENTS:

 - if T only return sounding events (pitches/chords) otherwise return the
   nearest rest or sounding event.
 - if T only return struck events i.e. ignore those that are tied to

RETURN VALUE:

 An event object.

SYNOPSIS:

```
(defmethod get-nearest-event ((sc slippery-chicken) reference-event
                              search-player &optional
                              (ignore-rests t)
                              (ignore-tied t))
```

29.32   slippery-chicken/get-note

[slippery-chicken] [Methods]

DESCRIPTION:

 Get a numbered event from a specified bar of a given player's part within a
 slippery-chicken object.

 NB: Slippery-chicken doesn't have 'note' and 'rest' classes, rather both of
     these are events. The nomenclature 'note' and 'rest' are thus used here
     and elsewhere merely for convenience, to distinguish between sounding
     and non-sounding events.

 See also rthm-seq-bar methods for accessing notes by other means.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar from which to get the note
   (counting from 1).
 - An integer that is the number of the note to get within that bar,
   counting tied notes (counting from 1). This can also be a list of numbers
   if accessing pitches in a chord (see below).
 - The ID of the player from whose part the note is to be retrieved.

OPTIONAL ARGUMENTS:

 - T or NIL to indicate whether, when accessing a pitch in a chord, to
   return the written or sounding pitch. T = written. Default = NIL.

RETURN VALUE:

 An event object.

EXAMPLE:

```
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((vn (violin :midi-channel 1))))
         :tempo-map '((1 (q 60)))
         :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
         :set-map '((1 (1)))
         :rthm-seq-palette '((1 ((((2 4) (e) e+e. 32 (32)))
                                  :pitch-seq-palette (((1) 2)))))
         :rthm-seq-map '((1 ((vn (1)))))))))
  (print (data (get-rest mini 1 2 'vn)))
  (print (data (get-note mini 1 2 'vn)))
  (print (data (get-note mini 1 '(2 1) 'vn)))
  (print (data (get-note mini 1 '(2 2) 'vn)))
  (print (is-tied-from (get-note mini 1 1 'vn))))

=>
32
"E."
C4
A4
T
```

SYNOPSIS:

```
(defmethod get-note ((sc slippery-chicken) bar-num note-num player
                     &optional written)
```

29.33   slippery-chicken/get-num-sections

[slippery-chicken] [Methods]

DESCRIPTION:

Return the number of sections in the given slippery-chicken object, as
defined in e.g. in the set-map. N.B. If the object has subsections these
are counted also, unless the optional argument is NIL.

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

 - count-subsections: if T (default) count all sections and subsections. If
 NIL just the top-level sections.

RETURN VALUE:

 An integer that is the number of section.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax (alto-sax :midi-channel 1))))
        :set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
        :set-map '((1 ((a (1 1 1 1))
                       (b (1 1 1 1))))
                   (2 (1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((a ((sax (1 1 1 1))))
                            (b ((sax (1 1 1 1))))))
                        (2 ((sax (1 1 1))))
                        (3 ((sax (1 1 1 1 1)))))))))
  (get-num-sections mini))

=> 4
```

SYNOPSIS:

```
(defmethod get-num-sections ((sc slippery-chicken)
                             &optional (count-subsections t))
```

29.34   slippery-chicken/get-phrases

[slippery-chicken] [Methods]

DESCRIPTION:

 This returns lists of events that make up phrases. Its notion of phrases
 is very simplistic but hopefully useful all the same: it's any sequence of
 sounding notes surrounded by rests.

ARGUMENTS:

```
- the slippery-chicken object
- a single symbol or list of symbols representing the players from the
ensemble. If nil all players will be processed.
```

OPTIONAL ARGUMENTS:

```
keyword arguments:
- :start-bar. An integer bar number where the process should start. If
NIL we'll default to 1. Default = NIL.
- :end-bar. An integer bar number where the process should end. If
NIL we'll default to the last bar. Default = NIL.
- :pad. If a phrase starts or ends mid-bar, pad the first bar with leading
rests and the last bar with trailing rests to ensure we have full bars.
```

RETURN VALUE:

```
A list of sublists, one for each requested player. Each player sublist
contains sublists also, with all the events in each phrase.
```

SYNOPSIS:

```
(defmethod get-phrases ((sc slippery-chicken) players
                        &key start-bar end-bar pad)
```

29.35   slippery-chicken/get-player

[slippery-chicken] [Methods]

DESCRIPTION:

```
Return the player object for the specified player.
```

ARGUMENTS:

```
- A slippery-chicken object.
- A player ID.
```

RETURN VALUE:

```
A player object.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((fl (flute :midi-channel 1))
                     (tp (b-flat-trumpet :midi-channel 2))
                     (vn (violin :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((fl (1 1 1 1 1))
                            (tp (1 1 1 1 1))
                            (vn (1 1 1 1 1))))))))
  (get-player mini 'vn))
=>
PLAYER: (id instrument-palette): SLIPPERY-CHICKEN-STANDARD-INSTRUMENT-PALETTE
doubles: NIL, cmn-staff-args: NIL, total-notes: 25, total-degrees: 3548,
total-duration: 20.000, total-bars: 5, tessitura: B4
LINKED-NAMED-OBJECT: previous: (TP), this: (VN), next: NIL
NAMED-OBJECT: id: VN, tag: NIL,
data:
INSTRUMENT: lowest-written: G3, highest-written: C7
...
```

SYNOPSIS:

```
(defmethod get-player ((sc slippery-chicken) player)
```

29.36   slippery-chicken/get-rest

[slippery-chicken] [Methods]

DESCRIPTION:

Retrieve the event object that contains the specified rest in a
slippery-chicken object by giving bar number, rest number and player.

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the number of the bar from which to retrieve the rest
  event object.
- An integer that is the number of the rest (not the number of the event)
  within that bar, counting from 1.
- The ID of the player from whose part to retrieve the rest object.

RETURN VALUE:

 An event object.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((vn (violin :midi-channel 1))
(vc (cello :midi-channel 2))))
:tempo-map '((1 (q 60)))
:set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((2 4) (e) e+e. 32 (32)))
:pitch-seq-palette (((1) 2)))))
:rthm-seq-map '((1 ((vn (1 1 1))
(vc (1 1 1)))))))))
(get-rest mini 2 1 'vc))

=>
EVENT: start-time: 2.000, end-time: 2.500,
duration-in-tempo: 0.500,
compound-duration-in-tempo: 0.500,
amplitude: 0.700
bar-num: 2, marks-before: NIL,
tempo-change: NIL
instrument-change: NIL
display-tempo: NIL, start-time-qtrs: 2.000,
midi-time-sig: NIL, midi-program-changes: NIL,
8va: 0
pitch-or-chord: NIL
written-pitch-or-chord: NIL
RHYTHM: value: 8.000, duration: 0.500, rq: 1/2, is-rest: T,
score-rthm: 8.0, undotted-value: 8, num-flags: 1, num-dots: 0,
is-tied-to: NIL, is-tied-from: NIL, compound-duration: 0.500,
is-grace-note: NIL, needs-new-note: NIL, beam: NIL, bracket: NIL,
rqq-note: NIL, rqq-info: NIL, marks: NIL, marks-in-part: NIL,
letter-value: 8, tuplet-scaler: 1, grace-note-duration: 0.05
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: E, tag: NIL,
data: E
```

SYNOPSIS:

```
(defmethod get-rest ((sc slippery-chicken) bar-num rest-num player)
```

29.37   slippery-chicken/get-section

[slippery-chicken] [Methods]

DESCRIPTION:

 Return the section object with the specified reference ID.

ARGUMENTS:

 - A slippery-chicken object.
 - A reference ID.

RETURN VALUE:

 A section object.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((sax (alto-sax :midi-channel 1))
(db (double-bass :midi-channel 2))))
:set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
:set-map '((1 (1 1 1 1 1))
(2 (1 1 1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((sax (1 1 1 1 1))
(db (1 1 1 1 1))))
(2 ((sax (1 1 1 1 1))
(db (1 1 1 1 1)))))))))
(get-section mini 2))

=>
SECTION:
RECURSIVE-ASSOC-LIST: recurse-simple-data: NIL
num-data: 2
linked: T
full-ref: (2)
ASSOC-LIST: warn-not-found NIL
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
BAR-HOLDER:
```

```
start-bar: 6
end-bar: 10
num-bars: 5
start-time: 20.0
end-time: 40.0
start-time-qtrs: 0
end-time-qtrs: 40.0
num-notes (attacked notes, not tied): 50
num-score-notes (tied notes counted separately): 50
num-rests: 0
duration-qtrs: 20.0
duration: 20.0 (20.000)
```

SYNOPSIS:

```
(defmethod get-section ((sc slippery-chicken) reference &optional (warn t))
```

29.38   slippery-chicken/get-section-bar-nums

[slippery-chicken] [Methods]

DATE:

```
 Mon 13 Jan 2020 15:23:17 GMT
```

DESCRIPTION:

```
 Get the start and end bar numbers of each section of a slippery chicken
 piece, handlily formatted in an assoc list.
```

ARGUMENTS:

```
 A slippery chicken object
```

OPTIONAL ARGUMENTS:

```
 Keyword args:
 :start - the first section to look at. (Default = 1)
 :end - the last section to look at. (Default = nil, look to the end.)
```

RETURN VALUE:

```
 An assoc list where each key is a section number and its data is a list
 containing the start and end bar numbers of that section.
```

EXAMPLE:

```
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((sax (alto-sax :midi-channel 1))))
         :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
         :set-map '((1 (1 1 1 1 1))
                    (2 (1 1 1 1 1))
                    (3 (1 1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                  :pitch-seq-palette ((1 2 3 4 5)))))
         :rthm-seq-map '((1 ((sax (1 1 1 1 1))))
                         (2 ((sax (1 1 1 1 1))))
                         (3 ((sax (1 1 1 1 1)))))))))
    (get-section-bar-nums mini))

=>
ASSOC-LIST: warn-not-found T
CIRCULAR-SCLIST: current 0
SCLIST: sclist-length: 3, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL,
                     this: NIL,
                     next: NIL
NAMED-OBJECT: id: SECTION-BAR-NUMS, tag: NIL,
data: (
NAMED-OBJECT: id: 1, tag: NIL,
data: (1 5)
**************


NAMED-OBJECT: id: 2, tag: NIL,
data: (6 10)
**************


NAMED-OBJECT: id: 3, tag: NIL,
data: (11 15)
**************
)
**************
```

SYNOPSIS:

```
(defmethod get-section-bar-nums ((sc slippery-chicken)
                                  &key (start 1) (end nil)
```

```
                                        (assoc-list-id 'section-bar-nums))
```

29.39   slippery-chicken/get-section-refs

[slippery-chicken] [Methods]

DATE:

 07-May-2012

DESCRIPTION:

 Return the reference IDs for all section and subsections of a given
 slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the first top-level section for which to return the
   reference IDs. As this number refers to the number of top-level sections
   only, any subsections will be contained in these and only count as 1.
 - An integer that is the number of consecutive sections to return section
   reference IDs.

RETURN VALUE:

 A list of lists containing the section reference IDs of the specified
 range in the slippery-chicken object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax (alto-sax :midi-channel 1))))
        :set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
        :set-map '((1 (1 1 1))
                   (2 (1 1 1))
                   (3 ((a (1 1 1))
                       (b ((x (1 1 1))
                           (y (1 1 1))))))
                   (4 ((a (1 1 1))
                       (b (1 1 1))
                       (c (1 1 1 1))))
                   (5 (1 1 1))
```

```
                    (6 (1 1 1))
                    (7 (1 1 1)))
       :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
       :rthm-seq-map '((1 ((sax (1 1 1))))
                       (2 ((sax (1 1 1))))
                       (3 ((a ((sax (1 1 1))))
                           (b ((x ((sax (1 1 1))))
                               (y ((sax (1 1 1))))))))
                       (4 ((a ((sax (1 1 1))))
                           (b ((sax (1 1 1))))
                           (c ((sax (1 1 1)))))))
                       (5 ((sax (1 1 1))))
                       (6 ((sax (1 1 1))))
                       (7 ((sax (1 1 1)))))))))
 (get-section-refs mini 2 4))

=> ((2) (3 A) (3 B X) (3 B Y) (4 A) (4 B) (4 C) (5))
```

SYNOPSIS:

```
(defmethod get-section-refs ((sc slippery-chicken) start-section num-sections)
```

29.40   slippery-chicken/get-sequenz-from-section

[slippery-chicken] [Methods]

DESCRIPTION:

 Get the sequenz object specified by the section, player and sequenz number
 given.

ARGUMENTS:

 - a slippery-chicken object
 - the player (symbol)
 - the sequenz number, counting from 1

RETURN VALUE:

 the respective sequenz object

SYNOPSIS:

```
(defmethod get-sequenz-from-section ((sc slippery-chicken)
                                      section-ref player-ref seq-num) ; 1-based
```

29.41   slippery-chicken/get-set-for-bar-num

[slippery-chicken] [Methods]

DATE:

 July 11th 2020, Heidhausen

DESCRIPTION:

 Get the complete-set object from the set-palette that was used in the
 generation of a specific bar.

ARGUMENTS:

 - the slippery-chicken object
 - the bar number (integer)

RETURN VALUE:

 a complete-set object or NIL if the bar doesn't exist or the first event of
 the bar has no set-ref slot value, for any of the players, for some reason
 (e.g. not initialized by make-slippery-chicken)

SYNOPSIS:

(defmethod get-set-for-bar-num ((sc slippery-chicken) bar-num)

29.42   slippery-chicken/get-starting-ins

[slippery-chicken] [Methods]

DESCRIPTION:

 Return the instrument object that is the first instrument object used by a
 specified player in a given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - A player ID.

RETURN VALUE:

 An instrument object.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                     (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                 (2 ((sax ((2 alto-sax) (5 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))
                        (2 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1)))))))))
  (get-starting-ins mini 'sax))

=>
INSTRUMENT: lowest-written: BF3, highest-written: FS6
lowest-sounding: CS3, highest-sounding: A5
starting-clef: TREBLE, clefs: (TREBLE), clefs-in-c: (TREBLE)
prefers-notes: NIL, midi-program: 66
transposition: EF, transposition-semitones: -9
score-write-in-c: NIL, score-write-bar-line: NIL
chords: NIL, chord-function: NIL,
total-bars: 5 total-notes: 25, total-duration: 20.000
total-degrees: 2920, microtones: T
missing-notes: (BQF3 BQF4), subset-id: NIL
staff-name: alto saxophone, staff-short-name: alt sax,

largest-fast-leap: 999, tessitura: BQF3
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: ALTO-SAX, tag: NIL,
data: NIL
```

SYNOPSIS:

```
(defmethod get-starting-ins ((sc slippery-chicken) player) ; symbol
```

29.43   slippery-chicken/get-tempo

[slippery-chicken] [Methods]

DESCRIPTION:

 Return the tempo object in effect for a specified bar of a given
 slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of a bar within that slippery-chicken
   object.

RETURN VALUE:

 A tempo object.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((sax (alto-sax :midi-channel 1))))
:tempo-map '((1 (q 60)) (5 (e 72)) (7 (q. 176 "prestissimo")))
:set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
:set-map '((1 (1 1 1 1 1 1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((sax (1 1 1 1 1 1 1 1)))))))))
(get-tempo mini 6))

=>
TEMPO: bpm: 72, beat: E, beat-value: 8.0, qtr-dur: 1.6666666
qtr-bpm: 36.0, usecs: 1666666, description: NIL
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: NIL, tag: NIL,
data: 72
```

SYNOPSIS:

```
(defmethod get-tempo ((sc slippery-chicken) bar-num)
```

29.44   slippery-chicken/get-time-sig

[slippery-chicken] [Methods]

DESCRIPTION:

 Get the time-sig object associate with a specified bar number in a given
 slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.
 - An integer that is the number of the bar for which to the time-sig object
   is to be returned. NB: Although this argument is listed as optional in
   the method definition (due to inheritance), it is actually required.

OPTIONAL ARGUMENTS:

 - (see above).

RETURN VALUE:

 A time-sig object.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((sax (alto-sax :midi-channel 1))))
:set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
:set-map '((1 (1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s)
((5 8) q e s s e)
((3 16) s e))
:pitch-seq-palette ((1 2 3 4 5 1 2 3 4 5 1
2)))))
:rthm-seq-map '((1 ((sax (1 1 1)))))))))
(get-time-sig mini 2))

=>
TIME-SIG: num: 5, denom: 8, duration: 2.5, compound: NIL, midi-clocks: 24, num-beats: 5
SCLIST: sclist-length: 2, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: "0508", tag: NIL,
data: (5 8)
```

SYNOPSIS:

```
(defmethod get-time-sig ((sc slippery-chicken) &optional bar-num)
```

29.45   slippery-chicken/get-transposition-at-bar

[slippery-chicken] [Methods]

DATE:

 24-Mar-2011

DESCRIPTION:

 Return the number of semitones difference between the sounding pitches and
 written pitches of a given player's part in a specified bar within a
 slippery-chicken object; e.g. bass clarinet = -14.

ARGUMENTS:

 - The ID of the player for whom the transposition value is sought.
 - An integer which is the number of the bar for which the transposition
   value is sought.
 - A slippery-chicken object.

RETURN VALUE:

 An integer.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax (alto-sax :midi-channel 1))))
        :set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1)))))))
  (get-transposition-at-bar 'sax 2 mini))

=> -9
```

SYNOPSIS:

```
(defmethod get-transposition-at-bar (player bar (sc slippery-chicken))
```

29.46   slippery-chicken/lp-display

[slippery-chicken] [Functions]

DESCRIPTION:

This function has exactly the same arguments as write-lp-data-for all and
only differs from that method in that after writing all the Lilypond text
files for the score, it calls Lilypond to render the PDF, which is then
opened automatically from within Lisp (if the value of
(get-sc-config 'lp-display-auto-open) is T).

In order to work properly, you'll need to make sure the value of the
'lilypond-command configuration parameter is set via (set-sc-config ...) to
the full path of your Lilypond command (not the app: it's usually
/path/to/Lilypond.app/Contents/Resources/bin/lilypond on OSX).

NB Only available with CCL and SBCL on OSX. With SBCL on OSX the output
from Lilypond is only printed once the process has exited, so it may take a
while until you see anything.

ARGUMENTS:

See write-lp-data-for-all

RETURN VALUE:

An integer: the shell's exit code for the PDF open command, usually 0 for
success. Second returned value is the patch to the PDF file.

SYNOPSIS:

(defun lp-display (&rest args)

29.47   slippery-chicken/make-slippery-chicken

[slippery-chicken] [Functions]

DESCRIPTION:

Make a slippery-chicken object using the specified data. This is the
function that will be used most often to "put it all together", and many of
its slots require full objects of other classes rather than just straight
data. These objects, such as rthm-seq-palette, rthm-seq-map etc, are also
documented in detail elsewhere in the robodoc and the user's manual.

NB Usually, when no set pitches are available for an instrument, an error
will be signalled. As of October 2018, you can instead allocate an 'empty
sequence' (rests) in such cases by setting the following before calling
make-slippery-chicken:
(set-sc-config 'pitch-seq-no-pitches-error nil)

ARGUMENTS:

- A symbol that is the name/ID of the object. The value passed to this
  argument will be made into a global variable, so that the newly created
  slippery-chicken object and the data it contains remain in memory and can
  be accessed and modified after the object is generated.

OPTIONAL ARGUMENTS:

keyword arguments:
NB: Although these arguments are technically optional, the slippery-chicken
    object will only be complete and make musical sense if many of the core
    elements are present.
- :title. A string that will be used as the title of the piece. The value
  given for this object will be used as both the header for the printable
  output as well as the base for any file names generated by the
  write-lp-data-for-all method. Default = "slippery-chicken-piece".
- :subtitle. A string that will be used as the subtitle of the piece. Works
  in Lilypond scores only.
- :instrument-palette. An instrument-palette object. This will be the
  palette of instrument objects available to the players of in the given
  slippery-chicken object's ensemble slot.
  Default = +slippery-chicken-standard-instrument-palette+.
- :ensemble. A recursive association list that will be used as the data to
  create an ensemble object populated with player objects within the
  slippery-chicken object. The format of this list will be a list of
  user-defined player IDs each coupled with a list of instrument object IDs
  from the current instrument-palette and various player object
  parameters. See the user's manual and robodoc entries on the ensemble and
  player classes for more detail.
- :set-palette. A recursive association list that will be used as the data
  to create a set-palette object within the slippery-chicken object. This
  object is where the collections of possible pitches for any given
  sequence are defined. The format of this list will be a list of IDs for
  each set of pitches, each coupled with a list of note-name symbols for
  the pitches that will be used to make that set. See the user's manual and
  the robodoc entry on the set-palette class for more detail.
- :set-map. A recursive association list that will be used as the data to
  create a set-map object within the slippery-chicken object. This is where

the order in which the pitch collections defined in the set-palette will be used in the piece. The format of this list will be a list of IDs from the given slippery-chicken object's structure coupled with a list of IDs from those given to the sets in the set-palette. There must be an equal number of sections in this list as there are in the rthm-seq-map, and they must have identical names. There must be an equal number of individual set IDs in each list paired with the section IDs as there are in the corresponding lists of the rthm-seq-map. See the user's manual and the robodoc entries for the set-map and sc-map for more detail.

- :rthm-seq-palette. A recursive association list that will be used as the data to create a rthm-seq-palette object within the slippery-chicken object. This object is where the collections of possible rhythm sequences for any given sequence in the piece are defined. This list will take the format of a list of IDs paired with a list of data for individual rthm-seq objects. These in turn will consist of one or more lists of rhythm data for rthm-seq-bar objects, as well pitch-seq-palettes and marks data for the individual rthm-seq objects to be created. See the user's manual as well as the robodoc entries for rthm-seq-palette, rthm-seq, rthm-seq-bar, rhythm, and pitch-seq-palette for more detail.

- :rthm-seq-map. A recursive association list that will be used as the data to create a rthm-seq-map object within the slippery-chicken object. This is where the order in which the rhythm sequences defined in the rthm-seq-palette will be used in the piece. It will take the format of a list of section IDs, of which there must be an equal number as are given in the set-map, each coupled with a list of player IDs, as defined in the ensemble slot of the given slippery-chicken object. The player IDs in turn are coupled with a list of IDs for rthm-seq objects, as defined in the rthm-seq-palette. Each of these lists must contain the same number of elements as are contained in each of the set-map sections. See the user's manual and robodoc entries for rthm-seq-map and sc-map for more details.

- :snd-output-dir. A string that will be used as the directory path for any output generated by clm-play in conjunction with sound files listed in the sndfile-palette (see below). Default = (get-sc-config 'default-dir).

- :sndfile-palette. A recursive association list that will be used as the data to create a sndfile-palette object within the slippery-chicken object. This is where the list is defined that contains all possible source sound files which may be used in conjunction with output generated by clm-play. This list will take the format of a list of IDs for sound-file groups, coupled with lists of file names and various other parameters associated with the sndfile class. The list of sound-file groups is followed by a list of directory paths where the given sound files are located and an optional list of file extensions. See the user's manual and the robodoc entries on sndfile-palette, sndfile, and clm-play for more detail.

- :tempo-map. A recursive association list that will be used as the data to create tempo objects within the slippery-chicken object. This is one of

two options for specifying the sequence of tempo changes for a given
piece (also see tempo-curve below). The format will be a list of integers
that are measure numbers within the piece, each coupled with tempo
indications in the form (beat-unit bpm). See the user's manual as well as
the robodoc entry for tempo-map for more detail. NB: This slot cannot be
used together with :tempo-curve.
- :tempo-curve. A list of data that will be used to create tempo objects
  within the slippery-chicken object, based on an interpolated list of
  break-point pairs. This is one of two options for specifying the
  sequence of tempo changes for a given piece (also see tempo-map above.)
  The first item in the list will be the number of bars between each new
  tempo object. The second item is the beat basis for the tempo objects
  made. The third and final argument is the list of break-point pairs, of
  which the first is a value on an arbitrary x-axis and the second is a
  number of beats-per-minute. See the user's manual and the robodoc entry
  for tempo-curve for more detail. NB: This slot cannot be used together
  with :tempo-map.
- :staff-groupings. A list of integers that indicate the placement of group
  brackets for the printable output. Each number represents a consecutive
  number of players, in the order they appear in the ensemble object, that
  will be included in each consecutive group. The sum of the numbers in
  this list must be equal to the number of players in the ensemble. See the
  user's manual for more detail.
- :instrument-change-map. A recursive association list that will be used as
  the data to create an instrument-change-map object within the
  slippery-chicken object. This will be used to indicate where those
  players in the ensemble that play multiple instruments will change
  instruments. The format will be a list of section IDs coupled with a list
  of player IDs, each of which in turn is coupled with a list of 2-item
  lists consisting of a sequence number paired with the ID (name) of one of
  the instrument objects assigned to that player in the ensemble
  object. See the user's manual and the robodoc entries for
  instrument-change-map for more detail.
- :set-limits-high. A recursive association list that will be used to limit
  the uppermost pitches of either the parts of individual players or of the
  entire ensemble. The format will be a list of player IDs, as defined in
  the ensemble object, each paired with a list of break-point pairs that
  consist of a value on an arbitrary x-axis paired with a note-name pitch
  symbol. These break-point envelopes are applied to the entire duration of
  the piece. See the user's manual for more detail.
- :set-limits-low. A recursive association list that will be used to limit
  the lowermost pitches of either the parts of individual players or of the
  entire ensemble. The format will be a list of player IDs, as defined in
  the ensemble object, each paired with a list of break-point pairs that
  consist of a value on an arbitrary x-axis paired with a note-name pitch
  symbol. These break-point envelopes are applied to the entire duration of

the piece. See the user's manual for more detail.
- :fast-leap-threshold. A number that is the longest duration of a note in
  seconds that can be followed by a leap of a large interval, as defined in
  the largest-fast-leaps slot of the instrument objects. Default = 0.125.
- :instruments-hierarchy. A list of player IDs from the given
  slippery-chicken object's ensemble that will specify the order in which
  slippery chicken's pitch selection algorithm will choose pitches for the
  instruments. By default (when NIL) this order follows the order in which
  the instrument objects appear in the ensemble object. See the user's
  manual for more detail. Default = NIL.
- :rehearsal-letters. A list of numbers that are measure numbers at which
  consecutive rehearsal letters will be placed. Since rehearsal letters are
  technically actually place on the right-hand bar line of the previous
  measure, measure 1 cannot be entered here. Slippery chicken automatically
  proceeds consecutively through the alphabet, so only numbers are required
  here. See the user's manual for more detail. If NIL, no rehearsal letters
  will be added to the score. Default = NIL.
- :avoid-melodic-octaves. T or NIL to indicate whether two linearly
  consecutive pitches in the part of a given player may be of the same
  pitch class but a different octave. T = avoid melodic octaves.
  Default = T.
- :instruments-write-bar-nums. A list of player IDs above whose parts in
  the score bar numbers should be written. If NIL, bar numbers will be
  written above the top player in each group. NB: This slot affects CMN
  output only. Default = NIL.
- :pitch-seq-index-scaler-min. A decimal number that affects the likelihood
  that slippery-chicken's pitch selection algorithm will choose pitches for
  an instrument that have also already been assigned to other players. In
  general terms, the higher this number is, the more likely it will be that
  instruments may be assigned the same pitches, though this will of course
  also be dependent on other factors, such as the characteristics of those
  instruments and the pitches in the current set. See the user's manual on
  pitches and the robodoc entries for pitch-seq for more detail.
  Default = 0.5.
- :bars-per-system-map. A list of 2-item lists, each of which consists of a
  measure number coupled with a number of measures to be placed in each
  system starting at that measure number. NB: This list only affects CMN
  output. See the user's manual on score layout for more details.
- :composer. A string that will be used for the composer portion of the
  header on the score's first page in LilyPond output. If NIL, no
  composer's name will appear in the score. Default = NIL.
- :year. A string that will be concatenated with the composer portion of the
  header on the score's first page in LilyPond output. If NIL, no
  year will appear in the score. Default = NIL.
- :rthm-seq-map-replacements. A list of lists in the format
  '(((1 2 va) 3 2) ((2 3 vn) 4 3)) that indicate changes to individual

elements of lists within the given rthm-seq-map object. Each such list
indicates a change, the first element of the list being the reference
into the rthm-seq-map (the vla player of section 1, subsection 2 in the
first example here), the second element is the nth of the data list for
this key to change, and the third is the new data. If NIL, no changes
will be made. See the robodoc entries for rthm-seq-map for more
detail. Default = NIL.
- :set-map-replacements. A list of lists in the format
  '((1 2 2) (3 3 1)) that indicate changes to individual elements of lists
  within the given set-map object. Each such list indicates a change, the
  first element of the list being the reference into the set-map (the
  section, followed by a subsection if any exist), the second element being
  the nth of the data list for to change, and the third being the new
  data. If NIL, no changes will be made. See the robodoc entries for sc-map
  for more detail. Default = NIL.
- :key-sig. A two-element list indicating starting key signature for the
  piece, e.g. '(ef minor). Usual note name symbols apply (e.g. ds = d
  sharp, bf - b flat). Implies nothing beyond the signature, i.e. no
  conformity to tonality expected. Default '(c major) i.e. no key
  signature.
- :transposition-curve. An envelope describing over the duration of the
  piece (but using any arbitrary x-axis range) what transpositions in
  semitones should be applied to the sets. Default = (0 0 100 0)
(- :warn-ties. This slot is now obsolete, but is left here for backwards
   compatibility with pieces composed with earlier versions of
   slippery-chicken. Default = T.)

RETURN VALUE:

 T

EXAMPLE:

```
;;; An example using all slots          ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :title "A Little Piece"
        :composer "Joe Green"
        :ensemble '(((fl ((flute piccolo) :midi-channel 1))
                     (cl (b-flat-clarinet :midi-channel 2))
                     (hn (french-horn :midi-channel 3))
                     (tp (b-flat-trumpet :midi-channel 4))
                     (vn (violin :midi-channel 5))
                     (va (viola :midi-channel 6))
                     (vc (cello :midi-channel 7))))
```

```
:set-palette '((1 ((fs2 b2 d4 a4 d5 e5 a5 d6)))
               (2 ((b2 fs2 d4 e4 a4 d5 e5 a5 d6)))
               (3 ((cs3 fs3 e4 a4 e5 a5 e6))))
:set-map '((1 (2 1 2 3 1 3 1))
           (2 (1 1 3 2 2 3 1))
           (3 (2 3 1 3 1 1 2)))
:rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                        :pitch-seq-palette ((1 2 3))))
                    (2 ((((4 4) (q) e (s) s h))
                        :pitch-seq-palette ((2 1 3))))
                    (3 ((((4 4) e (s) s h (q)))
                        :pitch-seq-palette ((3 2 1)))))
:rthm-seq-map '((1 ((fl (2 3 3 1 1 1 2))
                    (cl (3 2 1 1 2 1 3))
                    (hn (1 2 3 1 1 3 2))
                    (tp (2 1 1 3 3 2 1))
                    (vn (3 1 3 2 1 1 2))
                    (va (2 1 1 1 3 2 3))
                    (vc (1 2 3 1 3 2 1))))
                (2 ((fl (3 1 3 2 2 1 1))
                    (cl (1 1 2 3 1 3 2))
                    (hn (1 3 2 1 3 1 2))
                    (tp (1 1 1 3 3 2 2))
                    (vn (2 1 3 1 3 1 2))
                    (va (2 2 3 1 1 3 1))
                    (vc (1 3 1 2 2 1 3))))
                (3 ((fl (1 1 3 2 1 3 2))
                    (cl (2 1 2 3 3 1 1))
                    (hn (3 2 1 1 1 3 2))
                    (tp (3 3 1 1 2 1 2))
                    (vn (3 1 3 2 1 1 2))
                    (va (3 2 1 1 3 2 1))
                    (vc (1 3 2 1 2 3 1)))))
:snd-output-dir "/tmp"
:sndfile-palette '(((sndfile-grp-1
                     ((test-sndfile-1.aiff :start 0.021 :end 0.283)
                      (test-sndfile-2.aiff)
                      (test-sndfile-3.aiff)))
                    (sndfile-grp-2
                     ((test-sndfile-4.aiff :frequency 834)
                      (test-sndfile-5.aiff)
                      (test-sndfile-6.aiff))))
                   ("/path/to/test-sndfiles-dir-1"
                    "/path/to/test-sndfiles-dir-2"))
;; :tempo-map '((1 (q 84)) (9 (q 72))) ; ; ;
:tempo-curve '(5 q (0 40 25 60 50 80 75 100 100 120))
```

```
        :staff-groupings '(2 2 3)
        :instrument-change-map '((1 ((fl ((1 flute) (3 piccolo) (5 flute))))))
        :set-limits-low '((fl (0 c5 50 g5 100 c5))
                          (cl (0 c4 50 f4 100 c4))
                          (hn (0 f3 50 c4 100 f3))
                          (tp (0 c4 50 f4 100 c4))
                          (vn (0 e5 50 a5 100 e5))
                          (va (0 c3 50 f3 100 c3))
                          (vc (0 c2 50 f3 100 c2)))
        :set-limits-high '((fl (0 d6 50 a6 100 d6))
                           (cl (0 c5 50 a5 100 c5))
                           (hn (0 f4 50 c5 100 f4))
                           (tp (0 f5 50 c5 100 f5))
                           (vn (0 c6 50 e6 100 c6))
                           (va (0 g4 50 d5 100 g4))
                           (vc (0 c4 50 f4 100 c4)))
      :fast-leap-threshold 0.5
      :instruments-hierarchy '(fl vn cl tp va hn vc)
      :rehearsal-letters '(3 11 19)
      :avoid-melodic-octaves nil
      :instruments-write-bar-nums '(fl cl hn tp)
      :pitch-seq-index-scaler-min 0.1
      :bars-per-system-map '((1 1) (2 2) (3 3) (7 4) (11 5))
      :rthm-seq-map-replacements '(((1 va) 3 1) ((2 fl) 4 3))
      :set-map-replacements '((1 2 2) (3 3 1)))))
  (midi-play mini :midi-file "/tmp/mini.mid")
  (cmn-display mini)
  (write-lp-data-for-all mini))
```

SYNOPSIS:

```
(defun make-slippery-chicken (name &key
                                   rthm-seq-palette
                                   rthm-seq-map
                                   set-palette
                                   set-map
                                   sndfile-palette
                                   tempo-map
                                   tempo-curve
                                   (snd-output-dir (get-sc-config
                                                      'default-dir))
                                   instrument-change-map
                                   instruments-write-bar-nums
                                   bars-per-system-map
                                   staff-groupings
                                   rthm-seq-map-replacements
```

```
                                    set-map-replacements
                                    set-limits-low
                                    set-limits-high
                                    instrument-palette
                                    ensemble
                                    rehearsal-letters
                                    (fast-leap-threshold 0.125)
                                    instruments-hierarchy
                                    (title "slippery chicken piece")
                                    subtitle
                                    composer year
                                    (avoid-melodic-octaves t)
                                    (avoid-used-notes t)
                                    (pitch-seq-index-scaler-min 0.5)
                                    defer
                                    ;; MDE Mon Jul  2 16:08:42 2012
                                    (key-sig '(c major))
                                    ;; MDE Thu Sep  6 14:34:14 2018
                                    (transposition-curve '(0 0 100 0))
                                    (warn-ties t))
```

29.48   slippery-chicken/midi-play

[slippery-chicken] [Methods]

DESCRIPTION:

Generate a MIDI file from the data of the specified slippery-chicken
object. Note that for events that contain chords, the event's amplitude slot
will be used for all pitches unless each individual pitch object in the
chord has its amplitude slot set to a number (in which case the event's
amplitude slot will be ignored.)

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :midi-file. The name of the MIDI file to produce, including directory
   path and extension. Default is a filename extracted from the title of the
   sc piece, placed in the (get-sc-config 'default-dir) directory (default
   /tmp).
 - :voices. NIL or a list of player IDs indicating which of the players'

parts are to be included in the resulting MIDI file. If NIL, all players'
parts will be included. Default = NIL.
- :players. A synonym for voices. This is the preferred keyword as it aligns
  better with other classes, but :voices is retained for older code
  compatibility.
- :start-section. An integer that is the number of the first section for
  which the MIDI file is to be generated. Default = 1.
- :num-sections. An integer that is the number of sections to produce MIDI
  data for in the MIDI file. If NIL, all sections will be written.
  Default = NIL.
- :from-sequence. An integer that is the number of the sequence within the
  specified section from which to start generating MIDI data. NB: This
  argument can only be used when the num-sections = 1. Default = 1.
- :num-sequences. An integer that is the number of sequences for which MIDI
  data is to be generated in the resulting MIDI file, including the
  sequence specified in from-sequence. If NIL, all sequences will be
  written. NB: This argument can only be used when the num-sections = 1.
  Default = NIL.
- :update-amplitudes. T or NIL to indicate whether events should have their
  amplitude slots updated to ensure that any dynamic marks that have been
  added (and therefore amplitude slots updated as a side-effect) reflect
  the subsequent events' amplitudes also. This will override any amplitude
  slots that have been changed other than by (setf (amplitude ...)) or
  (add-mark ... 'some-dynamic). Also, if T, then hairpins (cresc, dim) will
  result in increasing/decreasing amplitudes over their extent (but this
  will only work if the amplitude/dynamic of the hairpins' starting and
  ending events are correct, and in that case intervening events'
  amplitudes will be overwritten no matter how they've been set; also this
  won't handle cases where there's a hairpin end and a new hairpin
  beginning on the same event). Default = T
- :force-velocity. Either: an integer between 0 and 127 (inclusive) that is
  the MIDI velocity value which will be given to all notes in the resulting
  MIDI file, or a function which takes an event object argument and
  returns a velocity from it (e.g. randomising the existing amplitude
  slightly). Default = NIL.
- :auto-open. Whether to open the MIDI file once written. Currently only
   available on OSX with SBCL or CCL. Uses the default app for MIDI files,
   as if opened with 'open' in the terminal. Default = Value of
   (get-sc-config 'midi-play-auto-open).
- :suffix. Add some text to the filename just before .mid?. Default = ""

RETURN VALUE:

Returns the path of the file written, as a string.

EXAMPLE:

```
;;; An example with some typical values for the keyword arguments. ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1 1 1))
                   (2 (1 1 1 1 1 1 1))
                   (3 (1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                                :pitch-seq-palette ((1 2 3))))
                            (2 ((((4 4) (q) e (s) s h))
                                :pitch-seq-palette ((1 2 3))))
                            (3 ((((4 4) e (s) s h (q)))
                                :pitch-seq-palette ((2 3 3))))
                            (4 ((((4 4) (s) s h (q) e))
                                :pitch-seq-palette ((3 1 2)))))
        :rthm-seq-map '((1 ((cl (1 2 1 2 1 2 1))
                            (hn (1 2 1 2 1 2 1))
                            (vc (1 2 1 2 1 2 1))))
                        (2 ((cl (3 4 3 4 3 4 3))
                            (hn (3 4 3 4 3 4 3))
                            (vc (3 4 3 4 3 4 3))))
                        (3 ((cl (1 2 1 2 1 2 1))
                            (hn (1 2 1 2 1 2 1))
                            (vc (1 2 1 2 1 2 1))))))))
  (midi-play mini
             :midi-file "/tmp/md-test.mid"
             :voices '(cl vc)
             :start-section 2))

;;; An example that passes a (lambda) function to :force-velocity. Usually, by
;;; default, event amplitudes between 0.0 and 1.0 will map onto MIDI velocities
;;; of 0 to 127. Here we map them to velocities of 0 to 100 instead.
(midi-play +jitterbug+ :force-velocity
           #'(lambda (event)
               (floor (* (amplitude event) 100))))
```

SYNOPSIS:

```
#+cm-2
(defmethod midi-play ((sc slippery-chicken)
                      &key
                        ;; no subsection refs: use from-sequence instead
```

```
(start-section 1)
;; these voices are used to get the actual sequence
;; orders i.e. each voice will be appended to <section>
;; when calling get-data.
;; if nil then all voices.
(voices nil)
;; MDE Thu Oct 18 17:07:57 2018 -- players is a more
;; appropriate keyword than voices but keep the latter
;; for historical code and make the two equivalent
(players nil)
;; add something to the filename just before .mid?
(suffix "")
(midi-file
 (format nil "~a~a~a.mid"
         (get-sc-config 'default-dir)
         (filename-from-title (title sc))
         suffix))
(from-sequence 1)
(num-sequences nil)
;; if nil we'll write all the sections
(num-sections nil)
;; MDE Mon Jun 13 12:30:55 2016
(update-amplitudes t)
;; MDE Tue Jun  4 19:06:11 2013
(auto-open (get-sc-config 'midi-play-auto-open))
;; if this is a 7-bit number we'll use this for all
;; notes
(force-velocity nil))
```

29.49   slippery-chicken/next-event

[slippery-chicken] [Methods]

DESCRIPTION:

Get the events from a specified player's part within a given
slippery-chicken object one after the other (e.g. in a loop). This method
must be called once with a bar number (or any other non-NIL value,
whereupon we start at bar 1) first in order to reset the counter; doing
this will return NIL. Once the counter has been reset, calling the method
without a bar number will return the events in sequence.

ARGUMENTS:

- A slippery-chicken object.
- A player ID.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to return only events that consist of
  attacked notes (i.e., no ties or rests). T = return only events with
  attacked notes. Default = NIL.
- T, NIL, or an integer to indicate the first bar from which events are to
  be retrieved. If T, the counter is reset to the first event of the
  player's part. This should be NIL after the first resetting call.
  Default = NIL
- NIL or an integer to indicate the last bar from which events are to be
  retrieved. If NIL, all events will be retrieved from the starting point
  to the last event in the given slippery-chicken object. Default = NIL.
  NB This arg should be used in the second call to next-event.
  E.g.: (next-event sc player nil start-bar)
         (loop for ne = (next-event sc player nil nil end-bar)
                while ne
                etc...

RETURN VALUE:

 An event object

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((cl (1 1 1))
                            (hn (1 1 1))
                            (vc (1 1 1)))))))))
  (next-event mini 'vc nil t)
  (loop for ne = (next-event mini 'vc)
     while ne
  collect (get-pitch-symbol ne)))

=> (E4 NIL F4 NIL G4 E4 NIL F4 NIL G4 E4 NIL F4 NIL G4)
```

SYNOPSIS:

```
(defmethod next-event ((sc slippery-chicken) player
                       &optional
                       (attacked-notes-only nil)
                       ;; could be a number too, whereupon it's the bar
                       ;; number to start at
                       (start-over nil)
                       (end-bar nil)) ; inclusive
```

29.50   slippery-chicken/num-bars

[slippery-chicken] [Methods]

DESCRIPTION:

 Return the number of bars in the piece.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 An integer that is the number of bars.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((fl (flute :midi-channel 1))
(tp (b-flat-trumpet :midi-channel 2))
(vn (violin :midi-channel 3))))
:set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
:set-map '((1 (1 1 1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((fl (1 1 1 1 1))
(tp (1 1 1 1 1))
(vn (1 1 1 1 1)))))))))
(num-bars mini))

=> 5
```

SYNOPSIS:

```
(defmethod num-bars ((sc slippery-chicken))
```

29.51   slippery-chicken/num-notes

[slippery-chicken] [Methods]

DESCRIPTION:

Returns the number of attacked notes in a given slippery-chicken object;
i.e., not including ties or rests.

ARGUMENTS:

- A slippery-chicken object.

RETURN VALUE:

An integer,

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((fl (flute :midi-channel 1))))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1)))
        :rthm-seq-palette '((1 ((((4 4) - e e e e - - e e e e -)))))
        :rthm-seq-map '((1 ((fl (1)))))))))
  (num-notes mini))

=> 8
```

SYNOPSIS:

```
(defmethod num-notes ((sc slippery-chicken))
```

29.52   slippery-chicken/num-players

[slippery-chicken] [Methods]

DESCRIPTION:

Get the number of players in a slippery chicken object's ensemble object.

ARGUMENTS:

```
- A slippery chicken object
```

RETURN VALUE:

```
- An integer.
```

SYNOPSIS:

```
(defmethod num-players ((sc slippery-chicken))
```

29.53 slippery-chicken/num-seqs

[slippery-chicken] [Methods]

DESCRIPTION:

```
Return the number of sequences (which may contain multiple bars) in a
specified section of a slippery-chicken object.

NB This will return the total number of seqs if there are sub-sections.
```

ARGUMENTS:

```
- A slippery-chicken object.
- The ID of the section for which to return the number of sequences.
```

RETURN VALUE:

```
An integer.
```

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax (alto-sax :midi-channel 1))))
        :set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
        :set-map '((1 (1 1 1 1))
                   (2 (1 1 1))
                   (3 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1))))
                        (2 ((sax (1 1 1))))
```

```
                             (3 ((sax (1 1 1 1 1)))))))))
  (num-seqs mini 2))
```

=> 3

SYNOPSIS:

```
(defmethod num-seqs ((sc slippery-chicken) section-ref)
```

29.54   slippery-chicken/player-ambitus

[slippery-chicken] [Methods]

DATE:

 February 9th 2017

DESCRIPTION:

 Find the highest and lowest events in the (given bars of the) piece,
 printing if requested.

ARGUMENTS:

 - the slippery-chicken object
 - the player ID (generally symbol)

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :written. T or NIL to indicate if the written (as opposed to sounding)
   pitches of the events should be handled/returned/printed. Default = NIL =
   sounding pitches
 - :print. Whether to print results to the Lisp interpreter.
 - :start-bar. The bar number to start processing. Default = 1.
 - :start-bar. The bar number to stop processing (inclusive). Default = NIL =
   the last bar.

RETURN VALUE:

 two values: the lowest event and the highest event.

SYNOPSIS:

```
(defmethod player-ambitus ((sc slippery-chicken) player &key written print
                                                (start-bar 1) end-bar)
```

29.55   slippery-chicken/player-doubles

[slippery-chicken] [Methods]

DATE:

 02-Apr-2012

DESCRIPTION:

 Boolean test to check whether a specified player plays more than one
 instrument.

ARGUMENTS:

 - A slippery-chicken object.
 - A player ID.

RETURN VALUE:

 T if the player has more than one instrument, otherwise NIL>

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((sax ((alto-sax tenor-sax) :midi-channel 1))
                     (db (double-bass :midi-channel 2))))
        :instrument-change-map '((1 ((sax ((1 alto-sax) (3 tenor-sax)))))
                                  (2 ((sax ((2 alto-sax) (5 tenor-sax))))))
        :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
        :set-map '((1 (1 1 1 1 1))
                   (2 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                :pitch-seq-palette ((1 2 3 4 5)))))
        :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                            (db (1 1 1 1 1))))
                         (2 ((sax (1 1 1 1 1))
                             (db (1 1 1 1 1)))))))))
  (player-doubles mini 'sax))

=> T
```

SYNOPSIS:

```
(defmethod player-doubles ((sc slippery-chicken) player)
```

29.56 slippery-chicken/players

[slippery-chicken] [Methods]

DESCRIPTION:

 Return a list of all player IDs from the given slippery-chicken object.

ARGUMENTS:

 - A slippery-chicken object.

RETURN VALUE:

 A list of player IDs.

EXAMPLE:

```
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((cl (b-flat-clarinet :midi-channel 1))
                     (hn (french-horn :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((cl (1 1 1))
                            (hn (1 1 1))
                            (vc (1 1 1))))))))
  (players mini))

=> (CL HN VC)
```

SYNOPSIS:

```
(defmethod players ((sc slippery-chicken))
```

29.57 slippery-chicken/players-ambitus

[slippery-chicken] [Methods]

DATE:

 August 5th 2017

DESCRIPTION:

Call the player-ambitus method for each player in the slippery-chicken
object.

ARGUMENTS:

see player-ambitus method

RETURN VALUE:

the lowest and highest events of all the players

SYNOPSIS:

```
(defmethod players-ambitus ((sc slippery-chicken)
                            &key written print (start-bar 1) end-bar)
```

29.58   slippery-chicken/rebar

[slippery-chicken] [Methods]

DESCRIPTION:

Go through the vertically simultaneous sequences in all players of the
given slippery-chicken object and rebar according to the first one that has
the least number of bars (but following the player hierarchy).

If rthm-seqs or sequenzes are created algorithmically and bundled into the
slippery-chicken piece slot artificially, bypassing the usual generation
structure, it might be difficult to end up with each instrument having the
same metric structure when combined vertically. This method goes through
the vertically combined sequences and rebars as described above.

NB: See documentation in piece class method. Don't confuse this method with
    the re-bar method.

NB: This method is used internally and not recommended for direct use.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

  - A list of player IDs from the given slippery-chicken object, ordered in
    terms of importance i.e. which instrument's bar structure should take
    precedence.
  - NB: The rebar-fun is not yet used.

RETURN VALUE:

 Always T.

SYNOPSIS:

```
(defmethod rebar ((sc slippery-chicken)
                  &optional instruments-hierarchy rebar-fun)
```

29.59   slippery-chicken/sc-init

[slippery-chicken] [Methods]

DESCRIPTION:

 Explicitly initialize the slippery-chicken object. This is usually called
 implicitly by initialize-instance (i.e. when you call
 make-slippery-chicken) but there could be circumstances (e.g. in subclasses
 of slippery-chicken) where you'd like to defer initialization and call this
 method explicitly instead. In that case set :defer to t when making the
 slippery-chicken object. This can also be called with using :defer, to

ARGUMENTS:

 - the slippery-chicken object

 OPTIONAL ARUGMENTS
 keyword arguments:
 - :regenerate-pitch-seq-map: the pitch-seq-map is generated here for each
 instrument using the pitch-seqs in the rthm-seq-palette. By setting this
 to T we can force regeneration (e.g. if the rthm-seq-palette has changed
 and we want to re-init the sc with different data). Default = T.

RETURN VALUE:

 the now fully initialized slippery-chicken object

EXAMPLE:

```
(let ((sc (make-slippery-chicken
           '+mini+
           :ensemble '(((vn (violin :midi-channel 1))))
           :set-palette '((1 ((gs4 af4 bf4))))
           :defer t
           :set-map '((1 (1 1 1)))
           :rthm-seq-palette '((1 ((((4 4) e e e e e e e e))
                                    :pitch-seq-palette ((1 2 1 1 1 1 1 1)))))
           :rthm-seq-map '((1 ((vn (1 1 1)))))))))
  (sc-init sc))
```

SYNOPSIS:

```
(defmethod sc-init ((sc slippery-chicken) &key (regenerate-pitch-seq-map t))
```

29.60   slippery-chicken/section-densities

[slippery-chicken] [Methods]

DATE:

 January 12th 2019, Heidhausen

DESCRIPTION:

 Calculate the densities of notes or rests in the sections of a
 slippery-chicken object. Density is the average number of notes/rests per
 bar across a section.

ARGUMENTS:

 - a slippery-chicken object

OPTIONAL ARGUMENTS:

 keyword arguments:
 - :slot. The section slot to use. 'num-score-notes is the default and makes
   sense in that ties indicate a higher level of density than just attacked
   notes by themselves, but further useful slots could be 'num-notes
   (attacks) or indeed 'num-rests
 - :percent. Whether to return values as percentages (where 0 would be the
   least dense section and 100 the most). If NIL the actual average notes per
   bar in each section would be returned. Default = T.

RETURN VALUE:

a list of density numbers, one per section/subsection.

SYNOPSIS:

```
(defmethod section-densities ((sc slippery-chicken)
                              &key (slot 'num-score-notes) (percent t))
```

29.61 slippery-chicken/shorten-large-fast-leaps

[slippery-chicken] [Methods]

DESCRIPTION:

Modify the pitches of each part in a slippery-chicken object to avoid large
melodic leaps at fast speeds, based on the largest-fast-leap slot of the
given instrument object and the fast-leap-threshold slot of the
slippery-chicken object.

This method is called automatically at init and as such will most likely
seldom need to be directly accessed by the user.

ARGUMENTS:

 - A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments:
 - :threshold. A number that is the maximum duration in seconds between two
   consecutive notes in the slippery-chicken object for which linear
   intervals greater than the number specified in the given instrument
   object's largest-fast-leap slot will be allowed. This value is taken from
   the fast-leap-threshold slot of the given slippery-chicken object by
   default.
 - :verbose. T or NIL to indicate whether to print feedback about the
   method's operations to the Lisp listener. T = print. Default = T.

RETURN VALUE:

 Always T

EXAMPLE:

```
(let ((mini
```

```
(make-slippery-chicken
'+mini+
:ensemble '(((vn (violin :midi-channel 1))))
:tempo-map '((1 (q 96)))
:set-palette '((1 ((g3 a5 b6))))
:set-map '((1 (1 1 1 1 1 1)))
:rthm-seq-palette '((1 ((((2 4) e s 32 64 64 e s 32 64 64))
:pitch-seq-palette ((1 5 1 5 1 5 1 5 1 5)))))
:rthm-seq-map '((1 ((vn (1 1 1 1 1 1))))))))))
(shorten-large-fast-leaps mini :threshold 0.25))

=>
******* section (1)
Getting notes for VN
Shortening short, fast leaps...
Shortened 23 large fast leaps
seq-num 0, VN, replacing B6 with G3
seq-num 0, VN, replacing B6 with G3
seq-num 0, VN, replacing B6 with G3
seq-num 0, VN, replacing B6 with G3
seq-num 0, VN, replacing G3 with B6
seq-num 0, VN, replacing G3 with B6
seq-num 0, VN, replacing G3 with B6
seq-num 1, VN, replacing G3 with B6
seq-num 1, VN, replacing B6 with G3
seq-num 1, VN, replacing B6 with G3
seq-num 1, VN, replacing B6 with G3
seq-num 1, VN, replacing B6 with G3
seq-num 1, VN, replacing G3 with B6
seq-num 1, VN, replacing G3 with B6
seq-num 1, VN, replacing G3 with B6
seq-num 2, VN, replacing G3 with B6
seq-num 2, VN, replacing B6 with G3
seq-num 2, VN, replacing B6 with G3
seq-num 2, VN, replacing B6 with G3
seq-num 2, VN, replacing B6 with G3
seq-num 2, VN, replacing G3 with B6
seq-num 2, VN, replacing G3 with B6
seq-num 2, VN, replacing G3 with B6
seq-num 3, VN, replacing G3 with B6
seq-num 3, VN, replacing B6 with G3
seq-num 3, VN, replacing B6 with G3
seq-num 3, VN, replacing B6 with G3
seq-num 3, VN, replacing B6 with G3
seq-num 3, VN, replacing G3 with B6
seq-num 3, VN, replacing G3 with B6
```

```
seq-num 3, VN, replacing G3 with B6
seq-num 4, VN, replacing G3 with B6
seq-num 4, VN, replacing B6 with G3
seq-num 4, VN, replacing B6 with G3
seq-num 4, VN, replacing B6 with G3
seq-num 4, VN, replacing B6 with G3
seq-num 4, VN, replacing G3 with B6
seq-num 4, VN, replacing G3 with B6
seq-num 4, VN, replacing G3 with B6
seq-num 5, VN, replacing G3 with B6
seq-num 5, VN, replacing B6 with G3
seq-num 5, VN, replacing B6 with G3
seq-num 5, VN, replacing B6 with G3
seq-num 5, VN, replacing B6 with G3
seq-num 5, VN, replacing G3 with B6
seq-num 5, VN, replacing G3 with B6
seq-num 5, VN, replacing G3 with B6
```

SYNOPSIS:

```
(defmethod shorten-large-fast-leaps ((sc slippery-chicken)
                                     &key threshold (verbose t))
```

29.62   slippery-chicken/statistics

[slippery-chicken] [Methods]

DESCRIPTION:

Print various information about the given slippery-chicken object to the
Lisp listener or other specified stream.

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

- NIL or a stream to which the information should be printed. If NIL, the
  method will not print the information to any stream. T = print to the
  Lisp listener. Default = T.

RETURN VALUE:

A number of formatted statistics about the given slippery-chicken object.

EXAMPLE:

```
(let ((mini
(make-slippery-chicken
'+mini+
:ensemble '(((sax (alto-sax :midi-channel 1))))
:set-palette '((1 ((e3 fs3 b3 cs4 fs4 gs4 ds5 f5))))
:set-map '((1 (1 1 1))
(2 (1 1 1)))
:rthm-seq-palette '((1 ((((4 4) h q e s s))
:pitch-seq-palette ((1 2 3 4 5)))))
:rthm-seq-map '((1 ((sax (1 1 1))))
(2 ((sax (1 1 1)))))))))
(statistics mini))

=>
+MINI+
"+MINI+-piece"
start-bar: 1
end-bar: 6
num-bars: 6
start-time: 0.0
end-time: 24.0
start-time-qtrs: 0
end-time-qtrs: 24.0
num-notes (attacked notes, not tied): 30
num-score-notes (tied notes counted separately): 30
num-rests: 0
duration-qtrs: 24.0
duration: 24.0 (24.000)
```

SYNOPSIS:

```
(defmethod statistics ((sc slippery-chicken) &optional (stream t))
```

29.63   slippery-chicken/transpose-events

[slippery-chicken] [Methods]

DESCRIPTION:

 Transpose the pitches of event objects in a specified region and a
 specified player's part.

ARGUMENTS:

- A slippery-chicken object.
- A player ID.
- An integer that is the first bar in which to transpose events.
- An integer that is the first event in that bar to transpose.
- An integer that is the last bar in which to transpose events.
- An integer that is the last event in that bar to transpose.
- A positive or negative number that is the number of semitones by which
  the pitches of the events in the specified region should be transposed.

OPTIONAL ARGUMENTS:

keyword argument:
- :destructively. T or NIL to indicate whether the pitches of the original
  event objects should be replaced. T = replace. Default = T.

RETURN VALUE:

Returns a list of events.

EXAMPLE:

```
;;; Print the pitches before and after applying the method ;
(let ((mini
        (make-slippery-chicken
         '+mini+
         :ensemble '(((sax (alto-sax :midi-channel 1))
                      (db (double-bass :midi-channel 2))))
         :set-palette '((1 ((c2 d2 g2 a2 e3 fs3 b3 cs4 fs4 gs4 ds5 f5 bf5))))
         :set-map '((1 (1 1 1 1 1)))
         :rthm-seq-palette '((1 ((((4 4) h q e s s))
                                 :pitch-seq-palette ((1 2 3 4 5)))))
         :rthm-seq-map '((1 ((sax (1 1 1 1 1))
                             (db (1 1 1 1 1)))))))))
  (print
   (loop for e in (get-events-from-to mini 'sax 3 2 5 3)
      collect (get-pitch-symbol e)))
  (transpose-events mini 'sax 3 2 5 3 11)
  (print
   (loop for e in (get-events-from-to mini 'sax 3 2 5 3)
      collect (get-pitch-symbol e))))

=>
(EF4 AF4 BF4 EF5 CS4 EF4 AF4 BF4 EF5 CS4 EF4 AF4)
(D5 G5 A5 D6 C5 D5 G5 A5 D6 C5 D5 G5)
```

SYNOPSIS:

```
(defmethod transpose-events ((sc slippery-chicken) player start-bar
                              start-event end-bar end-event semitones
                              &key (destructively t))
```

29.64   slippery-chicken/update-instrument-slots

[slippery-chicken] [Methods]

DATE:

 23rd August 2013

DESCRIPTION:

 This will go through the generated slippery-chicken object's bar structure
 and update each players' instruments' total-bars, total-notes,
 total-duration, and total-degrees slots, for statistical purposes only.
 This might be called by the user after performing one of the editing
 routines that deletes or changes notes, etc.

ARGUMENTS:

 - The slippery-chicken object.

RETURN VALUE:

 T

SYNOPSIS:

```
(defmethod update-instrument-slots ((sc slippery-chicken))
```

29.65   slippery-chicken/update-slots

[slippery-chicken] [Methods]

DESCRIPTION:

 Called by initialize-instance and others. Updates timings of events and
 statistics. Not generally called by the user but can be useful if
 post-generation editing has changed something fundamental to the structure.

ARGUMENTS:

- A slippery-chicken object

OPTIONAL ARGUMENTS:

- A tempo-map object (not just as a list). If not given, then the tempo-map
  from the slippery-chicken object will be used. Default = NIL.
- A number that is the start-time of the first event object in
  seconds. Default = 0.0.
- A number that is the start-time of the first event, in 'quarters' (for
  MIDI timing). Default = 0.0.
- A integer that is the number of the starting bar. Default = 1.
- The reference of the current section (for internal recursive use in the
  bar-holder class). Default = NIL.
- The nth sequence (for internal recursive use in the sequenz class).
  Default = NIL.
- T or NIL to indicate whether to print a warning to the Lisp listener when
  ties are being used at the beginning of a sequence. This argument is now
  obsolete and ignored, but remains for some backward compatibility.
  Default = T.

RETURN VALUE:

 The duration in seconds of the object; in this class: the whole generated
 piece.

EXAMPLE:

```
;;; Create a slippery-chicken object and print the start time of one of its
;;; events; call update-slots with a start time of 10.0 and print the start
;;; time of that same event to see the difference
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((vn (violin :midi-channel 1))))
        :tempo-map '((1 (q 60)))
        :set-palette '((1 ((c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((2 4) (s) (s) e e e))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((vn (1 1 1 1 1)))))))))
  (print (start-time (get-event mini 4 1 'vn)))
  (update-slots mini nil 10.0)
  (print (start-time (get-event mini 4 1 'vn))))

=
```

```
6.0
16.0
```

SYNOPSIS:

```
(defmethod update-slots ((sc slippery-chicken)
                         &optional
                           (tempo-map nil)
                           (start-time 0.0)
                           (start-time-qtrs 0.0)
                           (start-bar 1)
                           (current-section nil)
                           (nth nil)
                           (warn-ties t)
                           (update-write-bar-nums nil))
```

29.66   slippery-chicken/write-antescofo

[slippery-chicken] [Methods]

DESCRIPTION:

Write an antescofo˜ (Arshia Cont's/IRCAM's score follower MaxMSP external)
score file. This allows you to specify a single player to follow (for now:
no polyphonic following) and which players' events you'd like to be
triggered along with this, in the form of MIDI notes (sent via antescofo's
group action commands). Of course this doesn't imply that you have to use
MIDI with antescofo, rather, that you have something that picks up midi
note data and uses them (or ignores them) somehow or other. In any case,
the MIDI notes sent consist of four messages: the MIDI note (in midi cents
e.g. middle C = 6000), the velocity (taken as usual from the amplitude slot
of the event), the channel, and the duration in beats. Each may be preceded
by a delay, which will be relative to the previous NOTE or group MIDI note.

Rehearsal letters already in the slippery-chicken piece will automatically
be written into the antescofo˜ file as labels/cues. E.g. if you have
rehearsal letter A, it will show up in the antescofo˜ file as "letter-A".
Further labels/cues can be added to any slippery-chicken events and these
will also be written into the antescofo˜ file.

Action messages can also be added to events and thus written into the
antescofo˜ file. You can push as many messages as you want into this list;
they'll be reversed before writing out so that they occur in the order in
which you added them. If you prefer you can call the event method
add-antescofo-message. NB For the part we're following, we can add messages

to rests but if it turns out we added messages to rests in other players'
(i.e. group event parts) these won't be written to the antescofo~ file.

Do remember to set the instruments-hierarchy slot of the slippery-chicken
object so that the player you're going to follow is top of the list,
otherwise some strange event ordering might occur in the antescofo~ file.

Bear in mind that if you want to write antescofo~ files without having to
work within the usual slippery chicken workflow, you could generate events
by any method, then put them into rthm-seq-bar objects before then calling
bars-to-sc in order to create a slippery-chicken object by brute force (as
it were).

A note about grace notes: For the voice we're following, antescofo~
considers that grace notes are 'out of time' so have a duration of 0. But
for group notes, which we're triggering, they have to have some duration;
we default to the grace-note-duration of the event class (0.05 seconds by
default, which will be written as a fraction of the beat at the current
tempo, of course). Now this means our grace notes are not 'stealing' time
from the previous note, as they do in performed music. This is not ideal,
but if we've got a long group of grace notes, doing that might mean we end
up with a negative duration for the previous note. So we simply make the
grace notes short and allow/hope that the score follower catches up for us
on the next recognised note.

ARGUMENTS:

 - The slippery-chicken object
 - the player who we'll follow (single player for now, as a symbol)

OPTIONAL ARGUMENTS:

keyword arguments:
- :group-players (list of symbols). The players for whom midi-note events
  will be written in the antescofo file as part of a "group" action. If
  NIL, then we'll write all players' events except for the player we're
  following. NB There's no reason why we couldn't include the player we're
  following in these group commands (for unison playing between live and
  digital instruments perhaps). The easist way to write group events for
  all players is to write something like
  :group-players (players +your-sc-object+)
  Default = NIL.
- :bar-num-receiver. The MaxMSP receiver name to which bar numbers will be
  sent for display other other purposes. Default = "antescofo-bar-num"
- :midi-note-receiver. The MaxMSP receiver name to which midi notes will be
  sent. Default = "midi-note", so a typical group output event could be

```
   0.0 midi-note 6500 12 4 0.6666667
```
- :file. The name of the file to write. If NIL, then the file name will be
  created from the slippery-chicken title and placed in (get-sc-config
  'default-dir). Default = NIL.
- :group-duration-in-millisecs. Should we write the group notes (i.e. those
  we generate) as fractions of a beat (NIL) or have antescofo~ convert
  this to millisec duration according to the current tempo (T)? Default =
  NIL.
- :warn. Issue a warning when we write labels on group (rather than NOTE)
  events? (Because labels attached to group event notes can be written
  and read, but they won't show up in MaxMSP as cues which you can jump
  to.) Default = T.
- :messages-first. Write antescofo messages immediately before the note
  data sent to the midi-note-receiver (T), or after (NIL)? Default = NIL.
- :round-tempi. Round tempo values to the nearest integer? (If you don't do
  this, then MIDI playback could become out of synch with
  antescofo). Default = T.
- :include-program-changes. If T, we write any programme changes attached
  to an event as "midi-program-change <programme> <channel>". If this is a
  symbol or string then that ID will be used instead of
  "midi-program-change". (Symbols will be made lowercase.) Default = T.
- :makenote-compatible. If T, we'll write midi note information in the
  following order: midi note (rounded to nearest chromatic note i.e. no
  longer midi-cents), velocity, duration, channel. Default = NIL.

RETURN VALUE:

 The number of NOTE plus action events we've written. We also print to the
 terminal the number of events and actions separately, which should
 then correspond to what Antescofo~ prints when it loads the score in MaxMSP.

EXAMPLE:

```
;;; Follow the violin part and generate group events for all other parts ;
(let* ((mini
(make-slippery-chicken
'+mini+
:title "antescofo test"
:ensemble '(((vn (violin :midi-channel 1))
(va (viola :midi-channel 2))
(vc (cello :midi-channel 3))))
:set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
:set-map '((1 (1 1 1)))
:tempo-map '((1 60))
:rthm-seq-palette '((1 ((((4 4) { 3 tq tq tq } +q e (s) s)))))
:rthm-seq-map '((1 ((vn (1 1 1))
```

```
(va (1 1 1))
(vc (1 1 1))))))))
  ;; Adding a label (probably wouldn't need one in bar 1, but to illustrate) ;
(setf (asco-label (get-event mini 1 1 'vn)) "test-label")
  ;; start the (fictitious) vocoder when the first cello note in bar 2 is played ;
(push "max-receiver1 start-vocoder" (asco-msgs (get-event mini 2 1 'vc)))
(write-antescofo mini 'vn :file "/tmp/asco-test.txt"))

-->
******* section (1)
Getting notes for VN
Getting notes for VA
Getting notes for VC
Shortening short, fast leaps...
Shortened 0 large fast leaps
"/tmp/asco-test.txt"
Antescofo~ score written successfully with 15 events and 34 actions.
49


The generated file will begin something like this:

; antescofo~ score generated by slippery chicken version ;
; 1.0.4 (svn revision 4733 2014-01-15 11:27:10) ;
; at 12:02:06 on Thursday the 8th of May 2014 ;
BPM 60
antescofo-bar-num 1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;
NOTE 6200 0.6666667 test-label
group bar1.1 {
0.0 midi-note 6000 12 2 0.6666667
0.0 midi-note 6400 12 3 0.6666667
}
NOTE 6200 0.6666667
group bar1.2 {
0.0 midi-note 6000 12 2 0.6666667
0.0 midi-note 6400 12 3 0.6666667
}
NOTE 6200 1.6666667
group bar1.3 {
0.0 midi-note 6000 12 2 1.6666667
0.0 midi-note 6400 12 3 1.6666667
}
NOTE 6200 0.5
group bar1.4 {
0.0 midi-note 6000 12 2 0.5
0.0 midi-note 6400 12 3 0.5
}
```

```
NOTE 0 0.25
NOTE 6200 0.25
group bar1.5 {
0.0 midi-note 6000 12 2 0.25
0.0 midi-note 6400 12 3 0.25
antescofo-bar-num 2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;
}
NOTE 6200 0.6666667
group bar2.1 {
0.0 midi-note 6000 12 2 0.6666667
0.0 midi-note 6400 12 3 0.6666667
max-receiver1 start-vocoder
}
NOTE 6200 0.6666667
...
```

SYNOPSIS:

```
(defmethod write-antescofo ((sc slippery-chicken) follow-player
                            &key group-players file (warn t)
                              (group-duration-in-millisecs nil)
                             messages-first
                             ;; could be T, a string, or a symbol
                             ;; todo: include control changes also?
                             (include-program-changes t)
                             (round-tempi t)
                             makenote-compatible
                             (midi-note-receiver "midi-note")
                             (bar-num-receiver "antescofo-bar-num"))
```

29.67   slippery-chicken/write-lp-data-for-all

[slippery-chicken] [Methods]

DESCRIPTION:

 Generate all of the .ly files required by the LilyPond application for
 printable output from the musical data stored in the given slippery-chicken
 object.

 This method produces .ly files for the score as well as the parts for all
 individual players in the ensemble (unless otherwise specified by the
 user). The files are automatically named based on the value passed to the
 TITLE slot of the given slippery-chicken object.

NB: This method only produces the .ly files. These must be rendered by the
    LilyPond application separately for PDF output. See the slippery
    chicken installation web page and the manual page on Output for more
    detail. Bear in mind that SBCL and CCL users on OSX can use the
    lp-display macro to call Lilypond and display the resultant PDF
    automatically.

NB: Many of the arguments for this method pass their values directly to
    LilyPond parameters.

NB: Clefs added to grace-note events will not be rendered in Lilypond.

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

keyword arguments:
- :base-path. A string that is the directory path only for the resulting
  files. The method will automatically generate the file names and
  extensions. Default = (get-sc-config 'default-dir).
- :start-bar. An integer that is the first bar of the given
  slippery-chicken object for which output is to be generated. If NIL, the
  start-bar will be set to 1. Default = NIL.
- :end-bar. An integer that is the last bar of the given slippery-chicken
  object for which output is to be generated. If NIL, all bars after the
  start bar will be generated. Default = NIL.
- :start-bar-numbering: For bar counting in the score only: the bar number
  that the :start-bar will be counted as (integer). NIL = :start-bar.
  Default = NIL.
- :players. A list of player IDs or NIL to indicate which players' parts
  are to be generated and included in the resulting score. If NIL, all
  players' parts will be generated and included in the score. This can be
  handy, for example, for excluding the computer part of a piece for tape
  and instruments. Default = NIL.
- :respell-notes. NIL, T or a list to indicate whether the method should
  also call the respell-notes method on the given slippery-chicken object
  before generating the output to undertake enharmonic changes. If a list,
  then these are the specific enharmonic corrections to be undertaken. If
  this is T, the method will process all pitches for potential
  respelling. If NIL, no respelling will be undertaken. See the
  documentation for the respell-notes method for more. Default = NIL.
- :auto-clefs. T or NIL to indicate whether the auto-clefs method should be
  called to automatically place mid-measure clefs in the parts of
  instruments that use more than one clef. T = automatically place clefs.

```
    Default = T
```
- :in-c. T or NIL to indicate whether the full score is to contain written
  pitches or sounding pitches. NB: Some transposing C instruments still
  transpose at the octave in C scores, such as double-bass and piccolo.
  NB: Parts will always be transposed. T = sounding pitches. Default = NIL.
- :page-nums. T or NIL to indicate whether page numbers should
  automatically be added to each page (not including the start page) of the
  output. T = add page numbers. Default = T.
- :rehearsal-letters-font-size. A number that indicates the font size of
  rehearsal letters in LilyPond output. Default = 18.
- :rehearsal-letters-all-players. T or NIL to indicate whether rehearsal
  letters are to be placed in all parts generated. T = all parts.
  Default = T. NB: This must be set to T when the user would like the
  rehearsal letters in all individual LilyPond parts, but printing with CMN
  thereafter will result in rehearsal letters in all parts as well.
- :tempi-all-players. T or NIL to indicate whether tempo marks are to be
  placed in all parts generated. T = all parts. Default = T.
- :all-bar-nums. T o NIL to indicate whether the corresponding bar number
  should be printed above every measure in the score (not including
  multi-bar rests). T = add a bar number to every measure. Default = NIL.
- :paper. A string to indicate the paper size for LilyPond output. Only
  LilyPond's predefined paper sizes are valid here. According to the
  LilyPond manual, these include: "a4, letter, legal, and 11x17... Many
  more paper sizes are supported... For details, see scm/paper.scm, and
  search for the definition of paper-alist." NB: This argument will only
  adjust paper size, but not margins or line widths, which are adjusted
  using the arguments below. Default = "a4"
- :staff-size. An integer that indicates the size of the notes and staves
  in the resulting output. Standard for parts is 20. Default = 14.
- :group-barlines. T or NIL to indicate whether bar lines should be drawn
  through the whole staff group or just one staff. T = through the whole
  staff group. Default = T.
- :landscape. T or NIL to indicate whether the paper format should be
  landscape or portrait. T = landscape. NB: This argument will only adjust
  paper layout, but not margins or line widths, which are adjusted using
  the arguments below. Default = NIL.
- :barline-thickness. A number that is the relative thickness of the bar
  lines. Default = 0.5.
- :top-margin. A number that is the margin at the top of the page in
  millimeters. Default = 10.
- :bottom-margin. A number that is the margin at the bottom of the page in
  millimeters. Default = 10.
- :left-margin. A number that is the margin at the left of the page in
  millimeters. Default = 20.
- :line-width. A number that is the width of each line in centimeters.
  Default = 17.

- :page-turns. T or NIL to indicate if LilyPond should attempt to optimize
  page breaks for page turns in parts. T = optimize page breaks.
  Default = NIL.
- :min-page-turn. A two-item list indicating the minimum rest necessary for
  the method to automatically place a page turn, in a format similar to
  that of a time signature; i.e., '(2 1) would mean a minimum of 2 whole
  rests. Default = '(2 1))
- :use-custom-markup. T or NIL. Set to T when using a number of marks that
  are specific to LilyPond, such as 'bartok or any of the marks that use
  eps graphics files (whereupon those graphics files would need to be in
  the same folder as your lilypond files). Default = T.
- :lp-version. A string that will be added to each .ly file generated in
  conjunction with the LilyPond \version command. Default = "2.18.2"
- :process-event-fun. NIL or a user-defined function that will be applied
  to every event object in the given slippery-chicken object. If NIL, no
  processes will be applied. Default = NIL.
- :extend-hairpins. If you want hairpin (cresc/dim) to extend beyond the
   previous barline (or beyond the note) set to T. Default = NIL.
- :stemlet-length. NIL or a decimal number < 1.0 that indicates the scaled
  length of stems over rests in LilyPond output, should this feature be
  desired. 0.75 is a recommended value for this. NIL = no stems over
  rests. Default = NIL. NB: LilyPond can be instructed to extend beams over
  rests (without stemlets) simply by using the '-' in the definition of the
  rthm-seq-bar object, as is done with any other note; however,
  starting/ending a beam on a rest and then trying to generate a score with
  CMN will fail.
- :footer. A string to add to the footer of every score page. Default = NIL.
- :title. Usually T or NIL to include the title of the piece at the top of
  the score or not, but could also be a string to specify a new
  title. Default = T.
- :indent. T or NIL to indicate whether the first system in the score
  should be indented or not. Default = T.
- :force-bracket. T or NIL to indicate whether tuplet numbers on beams
  should be forced into a bracket. Default = NIL.
- :two-sided. A two-element list to set the inner and outer margins with
  layout for two-sided printing and binding. E.g. :two-sided '(30 20) means
  the left-hand margin will be 30mm and the right-hand will be 20mm. If
  set, this overrides the :left-margin and :line-width arguments.
  Default = NIL.
- :dummy-staves. The player name(s) as a symbol or a list for any parts
  that have been created merely as empty staves to add cross-staff
  notation.
- :between-system-space. The space in cm between systems (groups of
  staves). Default = NIL (= Lilypond's default distance).
- :staff-basic-distance. This controls the distance
  between staffs within a system. It's in Lilypond units (relative to

staff size). Setting to 15 creates a good distance between piano
staves. Default = NIL (= Lilypond's default distance).
- :fixed-width. Set to a rhythmic value (e.g. 1/32) in order to create
fixed width rhythmic spacing. This makes it easier, for some, to read
complex rhythms as you can see how long rhythms last by the horizontal
space they occupy. The shorter the value given the more space is
allocated, so experimentation might be necessary. Default = NIL (usual
engraver's spacing algorithm used, according to context).
- :accidental-style. String. See the lilypond documentation for details of
applicable styles and their results. Default = 'modern.

RETURN VALUE:

 The path of the main score file generated.

EXAMPLE:

```
;;; An example with values for the most frequently used arguments ;
(let ((mini
       (make-slippery-chicken
        '+mini+
        :ensemble '(((fl (flute :midi-channel 1))
                     (cl (b-flat-clarinet :midi-channel 2))
                     (vc (cello :midi-channel 3))))
        :staff-groupings '(2 1)
        :tempo-map '((1 (q 84)) (9 (q 72)))
        :set-palette '((1 ((f3 g3 a3 b3 c4 d4 e4 f4 g4 a4 b4 c5))))
        :set-map '((1 (1 1 1 1 1 1 1 1))
                   (2 (1 1 1 1 1 1 1 1))
                   (3 (1 1 1 1 1 1 1 1)))
        :rthm-seq-palette '((1 ((((4 4) h (q) e (s) s))
                                :pitch-seq-palette ((1 2 3))
                                :marks (bartok 1)))
                             (2 ((((4 4) (q) e (s) s h))
                                :pitch-seq-palette ((1 2 3)))))
        :rthm-seq-map '((1 ((fl (1 2 1 2 1 2 1 2))
                            (cl (1 2 1 2 1 2 1 2))
                            (vc (1 2 1 2 1 2 1 2))))
                         (2 ((fl (1 2 1 2 1 2 1 2))
                            (cl (1 2 1 2 1 2 1 2))
                            (vc (1 2 1 2 1 2 1 2))))
                         (3 ((fl (1 2 1 2 1 2 1 2))
                            (cl (1 2 1 2 1 2 1 2))
                            (vc (1 2 1 2 1 2 1 2)))))
        :rehearsal-letters '(3 11 19))))
  (write-lp-data-for-all mini
```

```
                                 :start-bar 7
                                 :end-bar 23
                                 :paper "letter"
                                 :landscape t
                                 :respell-notes nil
                                 :auto-clefs nil
                                 :staff-size 17
                                 :in-c nil
                                 :barline-thickness 3.7
                                 :top-margin 40
                                 :bottom-margin 60
                                 :left-margin 40
                                 :line-width 22
                                 :page-nums t
                                 :all-bar-nums t
                                 :use-custom-markup t
                                 :rehearsal-letters-font-size 24
                                 :lp-version "2.20.0"
                                 :group-barlines nil
                                 :page-turns t
                                 :players '(fl cl)
                                 :tempi-all-players t))

=> T
```

SYNOPSIS:

```
(defmethod write-lp-data-for-all
    ((sc slippery-chicken)
     &key
       (base-path (get-sc-config 'default-dir))
       start-bar end-bar (paper "a4") landscape
       ;; MDE Tue May 29 21:34:53 2012
       start-bar-numbering
       ;; if a list, then these are the enharmonic corrections
       (respell-notes t)
       ;; automatically add clefs to instruments who read more than one?
       (auto-clefs t)
       (staff-size 14)
       ;; parts will always be transposed but score can be in in C or not
       (in-c nil)
       (barline-thickness 0.5)
       (top-margin 10)                   ; mm
       (bottom-margin 10)                ; mm
       (left-margin 20)                  ; mm
       (line-width 17)                   ; cm
```

```
;; if not nil, then in cm
between-system-space
;; MDE Tue May  3 12:12:46 2016 -- this one controls the distance
;; between staffs within a system. It's in Lilypond units (relative to
;; staff size). Setting to 15 creates a good distance between piano
;; staves.
staff-basic-distance
(page-nums t)
;; print every bar number unless
;; multi-bar-rest?
(all-bar-nums nil)
;; this has to be T if we're going to get letters in the parts--but CMN
;; printing will have all parts all letters too thereafter
(rehearsal-letters-all-players t)
;; set to t if using bartok pizz and othersigns
(use-custom-markup t)
(rehearsal-letters-font-size 18)
;; "2.16.2") "2.14.2") ;"2.12.3") "2.17.95")
(lp-version "2.20.0")
;; 24.7.11 (Pula) barlines through whole staff group or just a stave
(group-barlines t)
;; 5.11.11 set to t if you want lilypond to optimize page breaks for
;; page turns in parts
(page-turns nil)
;; MDE Sat Mar 10 16:52:31 2012
(process-event-fun nil)
;; MDE Mon Apr 16 16:08:36 2012 -- added so that we can write a subset
;; of players into the score (e.g. leave out a computer part). If nil
;; all players will be written. NB the order of these will determine
;; the order in the score (overriding the ensemble order).
(players nil)
(footer nil)
;; minimum rest necessary to do a page turn; something like a time
;; signature e.g. (2 1) would mean we need a min. of 2 whole rests
(min-page-turn '(2 1))
;; MDE Tue May 29 22:58:25 2012
(stemlet-length nil)
;; MDE Thu Jan 9 09:20:33 2014 -- if you want hairpin (cresc/dim) to
;; extend beyond the note set to T
(extend-hairpins nil)
;; MDE Wed Jul  8 10:45:02 2015 -- tuplet numbers on beams don't usually
;; include a bracket but we can force this if we like
(force-bracket nil)
;; MDE Thu Mar 26 18:49:46 2015
(title t)
two-sided                          ; MDE Wed Oct 21 18:15:08 2015
```

```
;; MDE Sat Dec 24 14:54:34 2016 -- try also "dodecaphonic" if you want
;; accidentals before every pitch
(accidental-style "modern")
;; MDE Thu Mar 26 19:18:03 2015
(indent t)
;; MDE Thu Apr 28 15:17:36 2016 -- specify the player name(s) as a
;; symbol or in a list for any parts that have been created merely as
;; empty staves to add cross-staff notation.
dummy-staves
;; MDE Thu Feb  2 14:45:21 2017 -- fixed width rhythmic notation
fixed-width
;; sim to rehearsal letters
(tempi-all-players t))
```

29.68   slippery-chicken/write-xml

[slippery-chicken] [Methods]

DATE:

 March 17th 2017, Edinburgh

DESCRIPTION:

 Write a MusicXML file for import into Sibelius, Finale, Dorico, etc. All
 voices for the score and parts are written into one file. As it is assumed
 that editing will take place in the notation software of choice, there are
 only very basic arguments to this method, some of which may even be ignored
 upon import. For example, the :in-c keyword argument of other methods is
 missing here, as switching between sounding and transposing scores is
 assumed to be the job of the notation software used for MusicXML import.

 NB Sibelius 7.5 messes up some transposing instruments, getting octave
 transpositions wrong and adding generally unwanted key signatures.

ARGUMENTS:

 - the slippery-chicken object

OPTIONAL ARGUMENTS:

 keyword arguments
 - :staff-height. The suggested height of the staff, in millimetres. Default
   = 7.
 - :page-width. The page width in millimetres. Default = 210 = A4

- :page-height The page height in millimetres. Default = 297 = A4
- :tenths. A number representing tenths of interline staff space (positive
  or negative). Both integer and decimal values are allowed, such as 5 for a
  half space and 2.5 for a quarter space. Interline space is measured from
  the middle of a staff line. Distances in a MusicXML file are measured in
  tenths of staff space. Tenths are then scaled to millimeters within the
  scaling element, used in the defaults element at the start of a score. See
  https://usermanuals.musicxml.com/MusicXML/Content/EL-MusicXML-scaling.htm
  for more detail. Default = 40.
- :left-page-margins. A 4-element list specifying for left pages in the
  score the left, right, top, and bottom margins, all in
  millimetres. Default = '(20 20 20 20)
- :right-page-margins. Similar to :left-page-margins but for right pages.
- :players . Default = NIL = process all players.
- :start-bar . Default = NIL = 1.
- :end-bar . Default = NIL = the last bar in the slippery-chicken object.
- :respell-notes. NIL, T or a list to indicate whether the method should
  also call the respell-notes method on the given slippery-chicken object
  before generating the output to undertake enharmonic changes. If a list,
  then these are the specific enharmonic corrections to be undertaken. If
  this is T, the method will process all pitches for potential
  respelling. If NIL, no respelling will be undertaken. See the
  documentation for the respell-notes method for more detail. Default = T.
- :file. The full path of the file to write. Default will be derived from
  the title slot of the slippery-chicken object and will be written into the
  default directory e.g. (set-sc-config 'default-dir "~/Desktop")

RETURN VALUE:

 The file name written, as a string.

SYNOPSIS:

```
(defmethod write-xml ((sc slippery-chicken)
                    &key
                      (staff-height 7)  ; mm
                      (page-width 210)  ; mm = a4
                      (page-height 297) ; mm = a4
                      (tenths 40)       ; see below
                      ;; left right top bottom, all mm
                      (left-page-margins '(20 20 20 20))
                      (right-page-margins '(20 20 20 20))
                      players start-bar end-bar
                      (respell-notes t)
                      (file (format nil "~a_~a.xml"
                                    (get-sc-config 'default-dir)
```

```
(filename-from-title (title sc)))))
```